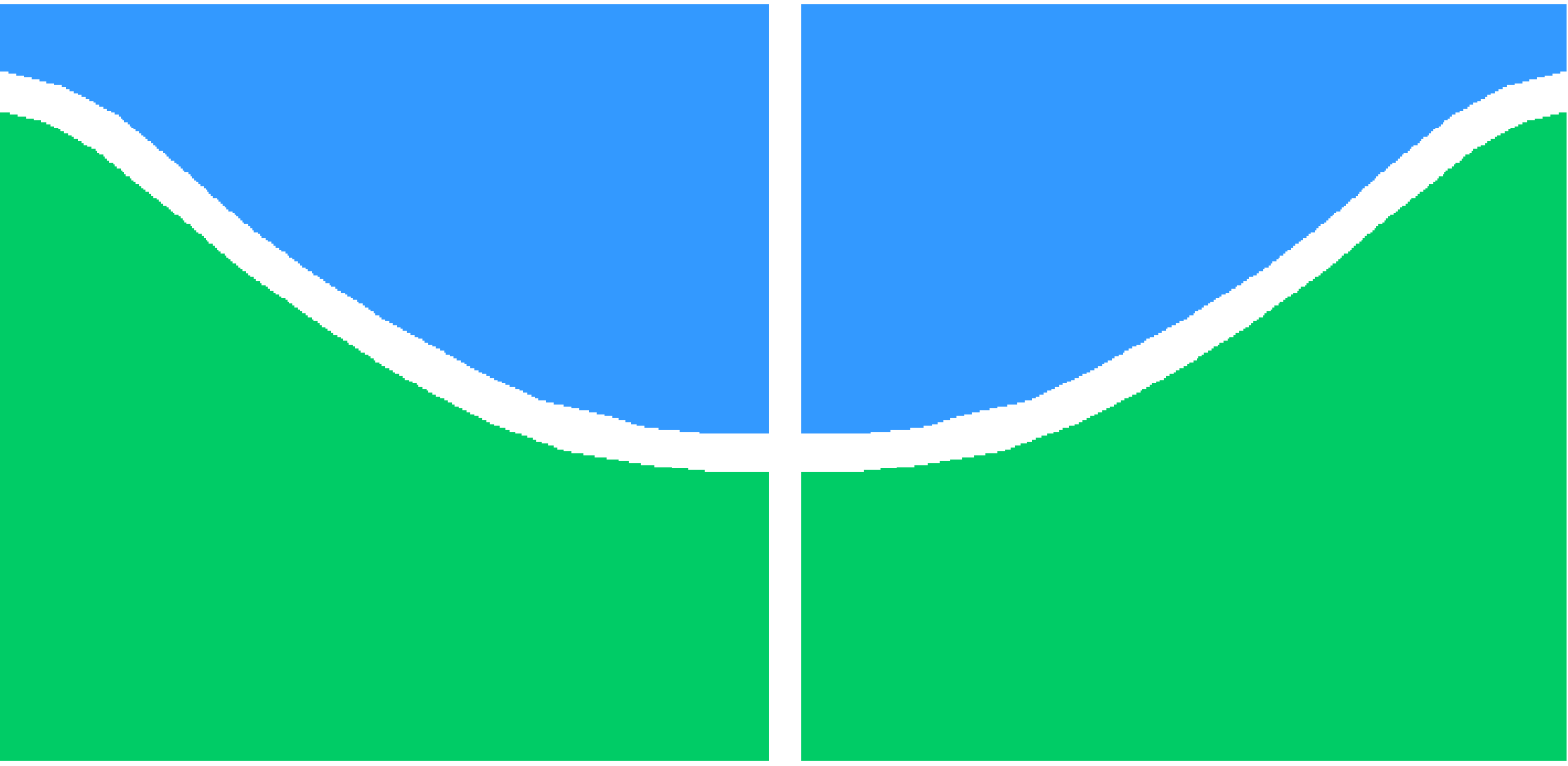

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Avaliação de métodos para redução do tempo de compilação de projetos escritos em C++

Autor: Macártur de Sousa Carvalho
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2015

Macártur de Sousa Carvalho

Avaliação de métodos para redução do tempo de compilação de projetos escritos em C++

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2015

Macártur de Sousa Carvalho
Avaliação de métodos para redução do tempo de compilação de projetos escritos em C++/ Macártur de Sousa Carvalho. – Brasília, DF, 2015-
140 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2015.

1. compilação. 2. preprocessador. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Avaliação de métodos para redução do tempo de compilação de projetos escritos em C++

CDU 02:141:005.6

Macártur de Sousa Carvalho

Avaliação de métodos para redução do tempo de compilação de projetos escritos em C++

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 14 de dezembro de 2015:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Paulo Roberto Miranda Meirelles
Membro Interno

Prof. Dr. Sérgio Antônio Andrade de Freitas
Membro Interno

Brasília, DF
2015

Resumo

Neste trabalho serão analisados métodos e ferramentas que proporcionam redução do tempo de compilação de programas escritos em C++, a fim de obter respostas mais rápidas às modificações no código e reduzir os gastos com recursos (humanos, de hardware e de tempo) utilizados no processo de compilação. Assim este trabalho consiste em uma análise de métodos e ferramentas que possa ser utilizado com o compilador na redução do tempo de compilação. Os métodos utilizados neste trabalho foram guardas de inclusão, declaração implícita de estruturas, (*forward declaration*), processamento paralelo com a ferramenta make, controle da ativação de flags de otimização, e ferramentas de auxílio a compilação(gold e ccache). Estes estudos foram aplicados em 3 sistemas operacionais distintos.

Palavras-chaves: compilação. C++. g++. Makefile. diretivas.

Abstract

This work will be analyzed methods and tools that provide reduction in compile-time programs written in C ++ in order to obtain faster responses to changes in the code and reduce spending on resources (human, hardware and time) used in the process compilation. So this work is an analysis of methods and tools that can be used with the compiler in reducing compile time. The methods used in this work were to include guards, implicit declaration of structures (textit forward declaration), parallel processing with make, control optimization flags activation, and compiling aid tools(gold and ccache). These studies were applied in three different operating systems.

Key-words: compiling. C++. g++. Makefile. directives.

Lista de ilustrações

Lista de tabelas

Listas de códigos

Lista de abreviaturas e siglas

DAG	Gráficos Aciclicos Dirigido
GIE	Guarda de Inclusão Externa
GII	Guarda de Inclusão Interna
GIEPO	Guarda de Inclusão Externa e Pragma Once
GIIPPO	Guarda de Inclusão Interna Primeiro que Pragma Once
JIT	*Just In Time*
MV	Máquina Virtual
PO	Pragma Once
POPGII	Pragma Once Primeiro que Guarda de Inclusão Interna
RGI	Redundância de Guarda de Inclusão
STL	*Standard Template Library*
TCC	Trabalho de Comclusão de Curso

Sumário

Parte I

Introdução

Introdução

Na computação, os primeiros computadores eletrônicos foram “engenhocas monstruosas”, que consumiam tanta energia quanto uma fábrica, e custavam, em 1940, milhares de dólares, com o poder computacional de uma calculadora moderna. Para os programadores que usavam estas máquinas a hora do computador era mais valiosas que a dele. E eles programavam em linguagem de máquina (SCOTT, 2009, pág.5).

Programadores ou desenvolvedores são pessoas capazes de criar instruções que podem ser executadas por um computador, escrevendo o código-fonte em alguma linguagem de programação. Finalizado o código-fonte, a construção de um programa que será executado em um computador pode ser feita por um compilador, programa capaz de fazer a transformação de código-fonte escrito na linguagem de programação escolhida para uma linguagem entendível pelo computador, que é o código de máquina (SCOTT, 2009, pág.5), a qual se trata de uma sequência de instruções binárias. Esta transformação recebe o nome de compilação (AHO; ULLMAN, 1977, pág.1).

Projetos de médio e grande porte gastam recursos e tempo a cada compilação, com perdas que dependem do hardware utilizado e do tamanho do projeto, sendo que a duração de um ciclo de compilação pode variar de alguns minutos até semanas (SCOTT, 2009, pág.5). Neste contexto, este trabalho realizar um levantamento de métodos e ferramentas para redução do tempo de compilação, uma vez que esta redução pode significar ganhos significativos de recursos humanos, financeiros e no cronograma de desenvolvimento de equipes e empresas.

Objetivos do Trabalho

Objetivo Geral

Analisar um conjunto de boas práticas e métodos para a redução do tempo de compilação de projetos escritos em C++ em diferentes ambientes, avaliando o impacto de cada estratégia. Serão selecionadas as melhores abordagens para os diferentes ambientes de desenvolvimento, tendo como meta conseguir uma redução mínima de 30% no tempo de compilação dos projetos analisados.

Objetivos Específicos

Para conseguir atingir o objetivo geral do trabalho foram listados alguns objetivos específicos:

1. Levantamento de métodos que envolvem a alteração de código fonte;
2. Levantamento de métodos que envolvem a parametrização de build;
3. Levantamento de ferramentas que podem auxiliar na compilação de projetos escritos em linguagem C++;
4. Elencar projetos de código livre para análise do tempo de compilação;
5. Coletar dados utilizando exemplos de uso de cada método aqui trabalhado, e avaliar o impacto de cada estratégia (método, ferramenta ou ambos) no tempo de compilação de cada projeto.
6. Elencar as melhores abordagems para cada método analisado.

Delimitação do Escopo

Para delimitar o escopo e abrangência deste trabalho, e tornar possível a análise de projetos reais, será utilizado como referência uma linguagem de programação compilada conhecida por ser utilizada em sistemas embarcados, sistemas operacionais, sistemas críticos e jogos, dentre outras aplicações, que é a C++, criada por Bjarne Stroustroup (STROUSTRUP, 2013).

Organização do Trabalho

O Capítulo 1 tem por objetivo introduzir o leitor nos conceitos abordados neste trabalho, descrevendo a diferença entre uma linguagem compilada e interpretada, o modelo *just-in-time*, máquinas virtuais, o processo de compilação e suas etapas e os métodos a serem trabalhados.

O Capítulo 2 descreve um fluxo de trabalho realizado para a primeira etapa deste trabalho, contendo projetos escolhidos, ferramentas utilizadas e condução dos método aplicado.

O Capítulo 3 apresenta os resultados alcançados depois da aplicação dos métodos, coleta de dados, bem como uma analise e seleção dos métodos que obtiveram melhores resultados em cada ambiente.

Por fim o Capítulo 4 aprensenta as considerações finais e quais os trabalhos futuros poderiam ser realizados e que não foram abordados neste trabalho.

Parte II

Desenvolvimento

1 Fundamentação Teórica

Este capitulo descreve os principais conceitos abordados neste trabalho, descrevendo a diferença entre linguagens compiladas e linguagens interpretadas, o modelo de execução *just in time*, máquinas virtuais, o processo de compilação e sobre os métodos aqui trabalhados.

1.1 Linguagens compiladas e linguagem interpretadas

Uma linguagem de programação é um método padronizado que possui um conjunto de instruções para comunicação entre o homem e o computador (DERSHEM; JIPPING, 1995), através de um conjunto de regras definidas tanto sintaticamente quanto sematicamente (FISCHER; GRODZINSKY, 1993). Com uma linguagem de programação é possível especificar precisamente um conjunto de regras sobre as quais dados serão armazenados ou transmitidos e criar algoritmos que representam operações ou resolvam problemas.

Para que um algoritmo escrito em uma linguagem de programação seja executado por um computador é necessário a criação de um arquivo de texto contendo um conjunto de palavras ou símbolos escritos de forma ordenada e de maneira lógica (AHO, 2008, pág. 3). Este arquivo é chamado de código fonte.

Antes da execução, a linguagem de programação escrita no código fonte deve ser convertida (através do processo de compilação), traduzida (através do processo de interpretação) ou convertida e traduzida (forma hibrida) em um código de máquina (COOPER, 2003, pág. 2).

1.1.1 Compilação

No processo de compilação a linguagem de programação contida no código fonte deve ser transformada em uma linguagem de máquina, a qual será posteriormente executada pelo processador (FISCHER; GRODZINSKY, 1993).

O processo de compilação requer a utilização de um compilador, o qual consiste em um programa ou conjunto de programas que tem como entrada o código fonte escrito em uma linguagem de programação e como saída, cria um programa sematicamente equivalente uma outra linguagem, chamado de código objeto (COOPER, 2003, pág. 2).

Após a compilação, a execução do código objeto criado a partir do código fonte de entrada deve ser requisitado pelo usuário para que o processador execute as instruções de máquina (SCOTT, 2009, pág. 17).

As linguagens que, em geral, são compiladas, tem melhor performance em relação as linguagens que, em geral, são interpretadas. Isso porque as decisões que são tomadas em tempo de compilação não são necessárias no tempo de execução (AHO, 2008, pág. 2). Por exemplo, se um compilador garantir que em uma variável `x` seja sempre alocada em uma mesma posição na memória, ele pode gerar instruções de máquina que acesse esta localização sempre que se referenciar a variável `x`. Por outro lado, um interpretador precisa procurar por `x` em uma tabela a cada acesso, a fim de determinar sua localização (FISCHER; GRODZINSKY, 1993, pág. 167).

Como exemplos de linguagens que, em geral, são compilada, temos Ada, ALGOL, BASIC, C, C++, COBOL, Cobra, Common Lisp, D, Delphi, Eiffel, Fortran, Objective-C, Pascal, Visual Basic, Visual Prolog.

1.1.2 Interpretação

No processo de interpretação, o código fonte é passado para um programa chamado interpretador, o qual é responsável pela leitura do código e pela tradução, em tempo real, deste código em instruções a serem executadas pelo sistema operacional ou pelo processador (FISCHER; GRODZINSKY, 1993). No caso de linguagens que são, em geral, interpretadas, na maioria dos casos são nomeadas como scripts, embora esta não seja uma regra geral.

Diferente do compilador, o interpretador faz a leitura do código fonte e executa as respectivas instruções em tempo real. Isso faz com que códigos interpretados tenham maior flexibilidade e um melhor diagnóstico (exibição de mensagens de erros) do que códigos compilados. A interpretação também pode lidar melhor com linguagens de programação nas quais características fundamentais, tais como o tamanho e o tipo das variáveis, podem depender dos dados de entrada (SCOTT, 2009, pág. 17).

Mesmo que uma linguagem seja projetada inicialmente para ser compilada é possível que exista interpretadores para estas linguagens, como é o caso do CINT[1] e CLING[2].Estes interpretadores podem ajudar no processo de desenvolvimento como descrito acima, no entanto existem limitações, como no caso do CINT[3]:

- não é possível definir um novo tipo de dado baseado em estrutura(*struct*) dentro de uma função;
- não suporta o tipo *unsigned long long*, interpretando-o como long long;
- o próprio **CINT** contém a definição do tipo bool, e carrega o mesmo na execução do programa;

[1] Interpretador de código C/C++ encontrado em <https://root.cern.ch/cint>.
[2] Interpretador de código C/C++ com back-end clang encontrado em <https://root.cern.ch/cling>.
[3] <https://root.cern.ch/viewvc/branches/v5-34-00-patches/cint/doc/limitati.txt>

- na declaração de ponteiro é necessário um espaço em branco antes e depois do '*';
- não aceita o operador de sequenciamento ',';
- um dos maiores problemas do **CINT** é a definição de macros: ele suporta apenas macros simples.

Sem o recurso da interpretação, seria uma tarefa deveras complexa implementar alguns recursos presentes em linguagens Lisp e Prolog onde, por exemplo, um programa pode escrever novas peças de si mesmo e executá-las em tempo real (SCOTT, 2009, pág. 17). Como exemplos de linguagens que, em geral, são interpretadas as linguagens: ActionScript, APL, ASP, BASIC, C#, CYBOL, Java, JavaScript, Lisp, Logo, Lua, PHP, Python, Ruby, Scheme, Smalltalk, VBScript.

1.1.3 Máquinas Virtuais

Para algumas linguagens existe o caso em que o compilador tem o papel de converter o código fonte em um *byte code* (SEBESTA, 2010, pág. 49). O *byte code* é uma linguagem de baixo nível similar a linguagem de máquina, que deve ser interpretada por um outro programa chamado Maquina Virtual (*Virtual Machine – VM*).

Esta estratégia mista de pré-compilar o código para uma linguagem intermediária e interpretá-la em uma máquina virtual é denominada estratégia híbrida.

Como exemplo de linguagens que utilizam, em geral, a estratégia híbrida, temos Java, Python, JRuby, Oxygene, Rhino, Nashorn, JGNAT, Jython, Rakudo Perl 6.

1.1.4 Just In Time

O termo *Just In Time* – JIT – veio de um novo modelo de negócio da indústria manufatureira: uma estratégia onde a produção ou compra era feita sobre demanda, ao invés da utilização de estoques (CARDOSO; AULER, 2014, pág. 177). Este modelo tem como vantagens menores custos com armazenamento, menos desperdícios, resposta mais rápida aos clientes e maior produção potencial.

Na compilação, esta analogia se adapta bem por que um compilador JIT não armazena os binários do programa no disco (o estoque), mas começa a compilação apenas de partes do programa necessárias durante a execução (FISCHER; GRODZINSKY, 1993, pág. 8).

Foram desenvolvidas, principalmente para linguagens que são, em geral, interpretadas, ferramentas que se valem do JIT para acelerar a interpretação e execução dos códigos-fonte (scripts) como, por exemplo, o PyPy para a linguagem Python.[4]

[4] <http://pypy.org/>

1.2 Processo de Compilação

1.2.1 Preprocessadores

Antes de um código-fonte passar pelo processo de compilação, pode ser necessária a execução de um programa, denominado preprocessador, que tem como responsabilidade preparar o código-fonte para a compilação.

Dentre as possíveis tarefas e características comuns a um preprocessador, podemos citar:

- *Processamento de macros*: um preprocessador pode permitir que um usuário defina macros que sejam abreviações para construções mais longas (AHO, 2008, pág. 8);
- *Inclusão de arquivos*: um preprocessador pode incluir arquivos cabeçalho no texto do programa. Por exemplo, o preprocessador faz com que o conteúdo de um arquivo externo seja transcrito no código-fonte no ponto onde existe a marcação para sua inclusão (FISCHER; GRODZINSKY, 1993, pág. 8);
- *Preprocessadores "racionais"*: este tipo de preprocessador é responsável por permitir a construção de macros utilizando condicionais *while* ou *if* mesmo em linguagens que não suportem tais estruturas (FISCHER; GRODZINSKY, 1993, pág. 8);
- *Extensões de linguagens*: são formas de conferir maior poder as linguagens através de macros embutidas. Por exemplo, Equel[5] é uma linguagem de interrogação embutida em C que permite a manipulação de banco de dados. Os enunciados começados com `##` são considerados pelo preprocessador como comandos de acesso ao banco de dados, os quais não fazem parte da linguagem C e, quando traduzidos, são convertidos em rotinas que tratam este acesso ao banco de dados (AHO, 2008, pág. 8).

1.2.2 Compilação

No processo de compilação tradicional, o compilador atua em duas fases principais: análise e síntese (SCOTT, 2009, pág. 26). A Figura 1 ilustra estas fases.

A fase de análise (ou *front-end*) de um compilador é a fase que tem como objetivo entender o código fonte e representá-lo em uma estrutura intermediária que facilite sua manipulação posterior. Esta fase é subdividida em análise léxica, análise sintática, análise semântica e geração de código intermediário (SCOTT, 2009, pág. 36).

A fase de síntese (ou *back-end*) de um compilador é a fase que tem como objetivo realizar a geração de código final, otimizando o código analisado na fase de sintase e

[5] <http://www.eecs.berkeley.edu/~wong/wong_pubs/wong46.pdf>. Acessado em 11 de junho de 2015.

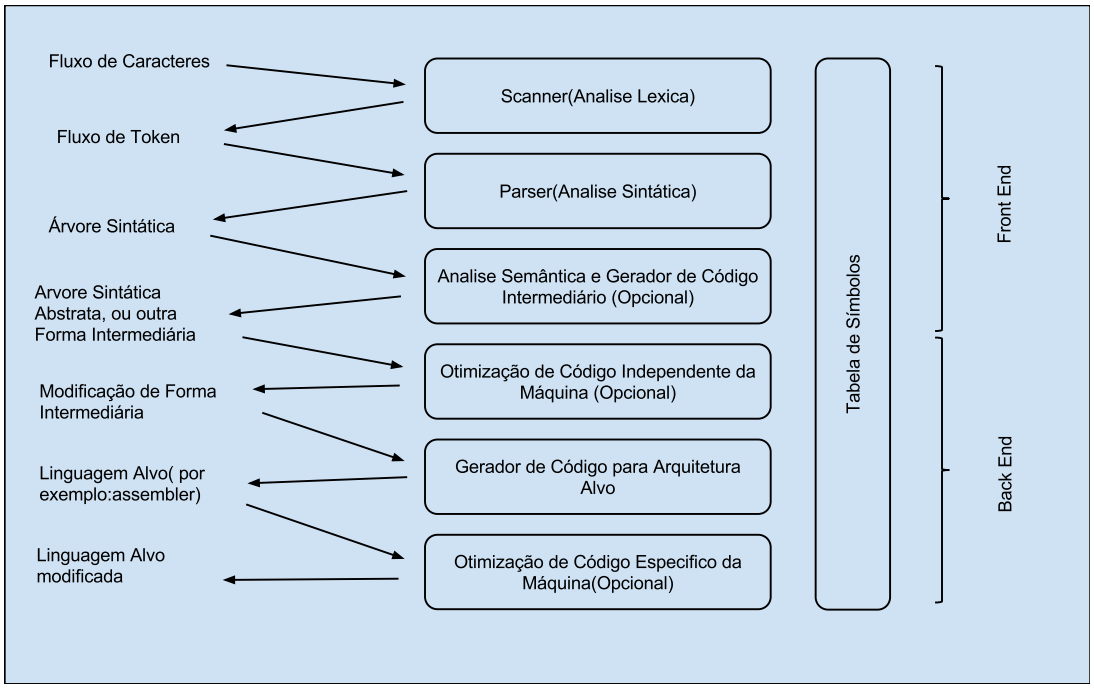

Figura 1: Fases da Compilação (SCOTT, 2009, pág. 26), com adaptações.

gerando um código sematicamente igual ao código fonte e com melhorias de performance e espaço. Esta fase é subdividida em otimização de código independente do alvo, geração de código alvo e otimização de código para o alvo específico (SCOTT, 2009, pág. 36).

1.2.3 Análise Léxica

A análise léxica é a primeira fase a ser executada pelo compilador (AHO; ULLMAN, 1972, pág. 59). A função do analisador léxico, também denominado *scanner*, é ler o código fonte, caractere a caractere, buscando a separação e a identificação dos elementos do programa, denominados símbolos léxicos ou *tokens* (PRICE; TOSCANO, 2000, pág. 195). Assim, é produzida uma sequência de *tokens* que será utilizada na análise sintática (AHO, 2008, pág. 38).

Esta fase também tem a importância de realizar a remoção de elementos "decorativos" do programa, tais como espaços, tabulações, caracteres de avanço e comentários (AHO; ULLMAN, 1972, pág. 59). Para auxiliar a construção deste analisador, estão disponíveis uma série de geradores automáticos de analisadores léxicos, cujo objetivo é reduzir o esforço de programação deste tipo de ferramenta, especificando-se apenas os *tokens* a serem reconhecidos (FISCHER; LEBLANC, 1991, pág. 50).

1.2.4 Análise Sintática

A análise sintática, ou análise gramatical, é o processo de determinar se uma cadeia de símbolos léxicos pode ser gerada por uma gramática pré-definida (AHO, 1986).

O analisador sintático é o responsável por verificar se os símbolos contidos no código fonte formam um programa válido ou não (DELAMARO, 2004, pág. 38).

A maioria dos métodos de análise sintática são de dois tipos, denominados *top-down* ou *bottom-up* (LEWIS; ROSENKRANTZ, 1978, pág. 227). Como indicado por seus nomes, os analisadores sintáticos *top-down* constroem árvores do topo para as folhas, enquanto o analisadores *bottom-up* começam das folhas e constroem a árvore de baixo para cima até chegar na raiz. Em ambos os casos são percorridas da esquerda para a direita, símbolo a símbolo. Estes dois tipos são utilizados devido seu potencial expressivo para descrever a maioria das construções sintáticas das linguagens de programação (DELAMARO, 2004, pág. 38). Para auxiliar na criação de analisadores sintáticos existem disponíveis uma série de geradores automáticos, como por exemplo, o *Flex*[6], o *Bison*[7] e o *JavaCC*[8] (ALBLAS; NYMEYER, 1996, pág. 30).

1.2.5 Análise Semântica

O analisador semântico tem como função prover métodos para que as estruturas construídas pelo analisador sintático possam ser avaliadas ou executadas (WATSON, 1989, pág. 337). Estas validações são feitas para assegurar que certos tipos de erros de programação sejam detectados e reportados. Os exemplos de verificação incluem declaração de tipo, declaração de funções, sobrecarga de funções, sobrecarga de operadores, verificação de fluxo de controle, verificação de operações lógicas e aritméticas válidas entre variáveis e a verificação de unicidade de variáveis em determinados escopos da linguagem (SCOTT, 2009, pág. 147).

1.2.6 Gerador de Código Intermediário

Após produzir uma árvore sintática sematicamente correta, o compilador é capaz de produzir uma linguagem de representação intermediária do código fonte. Uma linguagem intermediária está mais próxima de uma linguagem de objeto do que da linguagem do código fonte. No entanto, a linguagem intermediária permite uma manipulação mais fácil do que o código de máquina ou o *Assembly* (PYSTER, 1988, pág. 8).

Um tipo bem conhecido de linguagem intermediara é o código (ou sentença) de três endereços (CRESPO, 1998, pág. 247). Este código é uma sequência de enunciados na forma geral, apresentada no Código 1.1,

Código 1.1: Código de três endereços

```
1    x:=y op z
```

[6] <http://dinosaur.compilertools.net/>. Acessado no dia 20/06/2015.
[7] <http://www.gnu.org/software/bison/> Acessado no dia 20/06/2015.
[8] <https://javacc.java.net/>. Acessado no dia 20/06/2015.

onde `x`, `y`, `z` são nomes, constantes ou objetos de dados temporários criados pelo compilador e `op` representa um operador qualquer, tal como um operador de ponto fixo, flutuante ou um operador lógico sobre dados booleanos (SCOTT, 2009, pág. 201). Uma forma prática de representar sentenças de três endereços é o uso de quádrupla (operador, argumento 1, argumento 2 e resultado). Este esquema de representação de código intermediário é o preferido por diversos compiladores, principalmente por aqueles que executam extensivas otimizações de código, uma vez que o código intermediário pode ser manipulado mais facilmente (AHO; ULLMAN, 1977, pág. 604). Além destas representações existem outras como as árvores, grafos acíclicos dirigidos (DAG) e a notação polonesa (MUCHNICK, 1977, pág. 96).

Para garantir que um código tenha o seu desempenho ampliado e utilize menos espaço em disco, a fase de otimização de código busca examinar estrategicamente o código intermediário produzido na fase anterior e, com o uso de técnicas de otimização, produzir um código mais eficiente (SORENSON, 1989, pág. 796) . A otimização de código pode ocorrer em duas etapas: uma após a geração de código intermediário e a outra depois da geração do código para a máquina alvo (SCOTT, 2009, pág. 34). Para a execução desta etapa são utilizadas técnicas para detectar padrões dentro de um código produzido e substituí-los por códigos mais eficientes (AHO; ULLMAN, 1977, pág. 604). Entre as técnicas usadas realizadas estão a substituição de expressões que podem ser calculadas durante o tempo de compilação, movimentação de código, eliminação de sub-expressões redundantes, desmembramento de laços, eliminação de variáveis de indução, substituição de multiplicação pelo *shift* binário, redução da quantidade de laços, entre outras (SORENSON, 1989, pág. 796).

1.2.7 Geração de Código

Gerador de código é a parte ou componente do compilador responsável por realizar o mapeamento da linguagem intermediária otimizada para a linguagem alvo. Caso a linguagem alvo seja um código de máquina, registradores ou localização de memoria são selecionados para armazenar valores das variáveis usadas no programa. Então, o código intermediário é traduzido em sequência para instruções de máquina que realizarão as operações (AHO, 2008, pág. 10). Mesmo que o código traduzido seja otimizado, após a tradução para a linguagem alvo é possível que novas otimizações sejam feitas, como a otimização específica apresentada no Código 1.2.

Código 1.2: Exemplos de otimização específica

```
1 Propagacao de constante
2
3 r2 := 4                  r2 := 4                    r3 := r1 + 4
4 r3 := r1 + r2   vira     r3 := r1 + 4  e entao      r2 := ...
5 r2 := ...                r2 := ...
```

```
6
7  Duplicacao de Constantes
8
9  r2 := 3 x 2 vira r2 := 6
10
11 Eliminacao de Atribuicao e uso de variavel
12
13 r2 := r1 + 5                r2 := r1 + 5
14 i  := r2            vira    i   := r2
15 r3 := i                     r3 := r2 x 3
16 r3 := r3 x 3
```

1.2.8 Assembly

Segundo Michael L. Scott, um código de máquina é uma sequência de *bits* que corresponde a uma instrução executada por um processador, realizando operações de adições, comparações, movimento de informação de uma localização da memória, entre outros. Detalhar instruções de máquina a nível de *bits* é uma tarefa trabalhosa (SCOTT, 2009, pág. 5).

Um programa capaz de realizar o cálculo do máximo divisor comum através do algoritmo de Euclides estendido pode ser representado em código de máquina usando notação hexadecimal para a representação dos *bits*, conforme apresentado no Código 1.3 (SCOTT, 2009, pág. 5).

Código 1.3: Algoritmo de Euclides Estendido, com adaptações

```
1  55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 a2 00
2  00 00 39 c3  74 10 8d b6  00 00 00 00  39 c3 7e 13  29 c3 39 c3
3  75 f6 89 c1  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

Para facilitar a comunicação entre uma linguagem composta apenas por *bits* e uma linguagem entendida por um desenvolvedor foi necessário a criação de uma linguagem que permitisse que as operações em *bits* fossem representadas por abreviações ou símbolos que facilitassem o entendimento por parte do programador. Assembly é a linguagem escolhida para estas representações, onde cada instrução do processador foi mapeada em um mnemônico, representados geralmente por acrônimos do inglês (por exemplo, `mov` representa 'mover', `rep` representa repetição e assim por diante) (SCOTT, 2009, pág. 5).

O mesmo programa mostrado no Código 1.3 pode ser representado em *assembly*, conforme ilustrado no Código 1.4 (SCOTT, 2009, pág. 5).

Código 1.4: Algoritmo de Euclides estendido em Assembly , com adaptações

```
1      pushl %ebp                jle   D
2      mov   %esp, %ebp          subl  %eax, %ebx
```

```
3       pushl %ebx              B: cmpl  %eax, %ebx
4       subl  $4, %esp             jne   A
5       andl  $-16, %esp        C: call  %ebx, (%esp)
6       call  getint               call  putint
7       movl  %eax, %ebx           movl  -4(%ebp), %ebx
8       call  getint               leave
9       cmpl  %eax, %ebx           ret
10      je    C                 D: subl  %eax, %eax
11   A: cmpl  %eax, %ebx           jmp   B
```

Para converter um programa de *assembly* para um código de máquina é necessário um montador, denominado *Assembler* ou montador. O *assembly* é um programa que realiza a parametrização das instruções da linguagem Assembly para os *bits* correspondentes da linguagem de máquina (também chamado de *opcode*) (SCOTT, 2009, pág. 6).

1.2.9 *Linking*

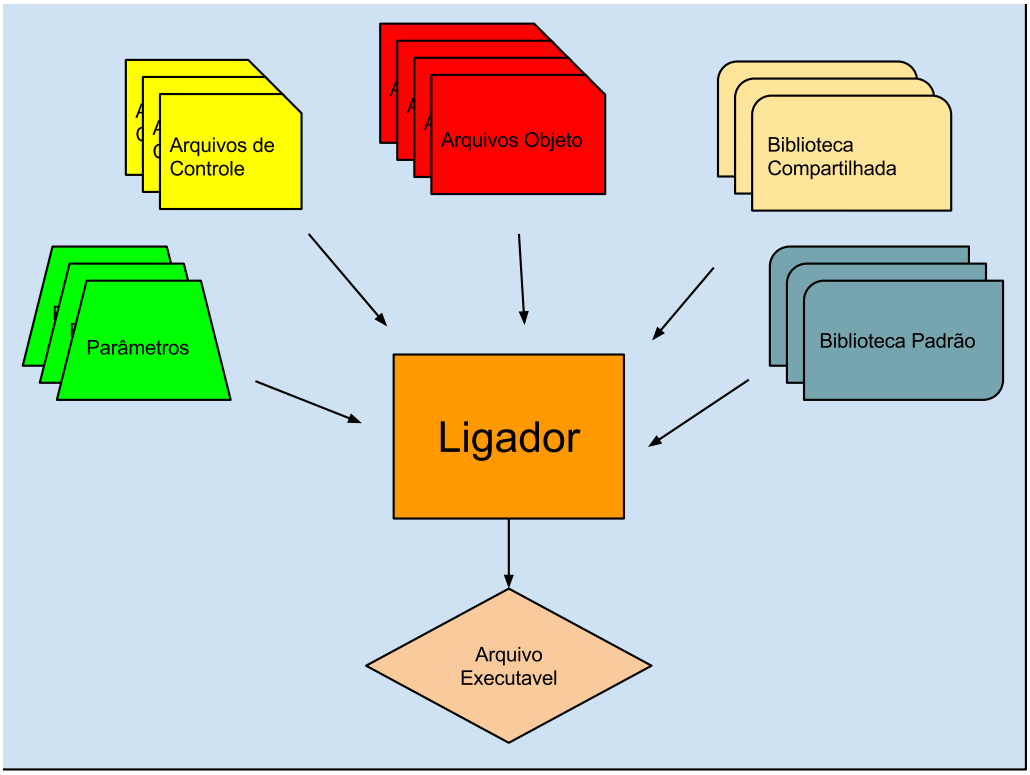

Figura 2: Ligação entre objetos (LEVINE, 2000, pág. 7), com adaptações.

Um compilador é capaz de realizar a compilação de partes de programas e gerar objetos que podem ser combinados para formar um programa executável. *Linker* (ou Ligador) é um programa capaz de realizar a junção de objetos gerados por um compilador, ligando nomes abstratos em nomes mais concretos, permitindo o desenvolvedor escrever códigos usando nomes mais próximos da sua realizade (LEVINE, 2000).

Basicamente o Linker liga um nome ou símbolo escrito por um desenvolvedor e referencia a localização do inicio do código que possui a função ou o dado estático. Como no exemplo descrito por John Levine, o *linker* pega um referência escrita como `getline` e vincula a "uma localização de 612 *bytes* no inicio de um código executável do módulo `iosys`" (LEVINE, 2000, pág. 7).

A fase de link-edição tem dois passos. No primeiro o Linker recebe como entrada um conjunto de arquivos objetos, bibliotecas e parâmetros e produz como resultado um arquivo de saída, como mostrado na Figura 2 (LEVINE, 2000, pág. 13). Neste passo é criada uma tabela com todos os segmentos definidos nos arquivos fontes e uma tabela de símbolos importados ou exportados. Depois o *Linker* atribui uma localização numérica para cada símbolo, determinando o tamanho e a localização dos segmentos no espaço de endereço final. O segundo passo usa as informações do primeiro para controlar a link-edição, ajustando os endereços de memória do código e escrevendo os endereços de código realocado no arquivo de saída (programa executável).

1.2.10 Bibliotecas

Após algum tempo utilizando compiladores e *linkers*, muitos desenvolvedores perceberam que poderiam economizar tempo e esforço reutilizando pedaços de códigos escritos em outros programas. Para evitar a cópia de arquivos entre projetos surgiram as chamadas bibliotecas (LEVINE, 2000, pág. 227).

Biblioteca é um conjunto de códigos compilados que podem ser incorporados a um ou mais de um projetos, como mostrado nas Figura 3, e Figura 4. A organização de códigos em bibliotecas permite que programas sejam mais modulares, mais rápidos de recompilar e mais fáceis de manter (WHEELER, 2015a).

As bibliotecas podem ser divididas em três tipos: estáticas, dinâmicas e compartilhadas (WHEELER, 2015a).

1.2.11 Bibliotecas Estáticas

Bibliotecas estáticas (que normalmente são nomeadas com o sufixo `.a`) são módulos de programas compilados separadamente, que podem ser utilizados na construção de um programa executável. Assim, após a etapa de compilação de um projeto, o *linker* faz a ligação entre as bibliotecas estáticas como mostrado na Figura 3 (STEVANOVIC, 2014, pág. 54).

Estas bibliotecas são mais difíceis de se manter, pois a cada atualização de uma biblioteca estática todos os projetos dependentes da mesma devem ser recompilados. Outra dificuldade é que um o binário final do projeto fica maior em relação aos outros tipos de bibliotecas (pois ele incorpora uma cópia de cada biblioteca estática utilizada) e pode con-

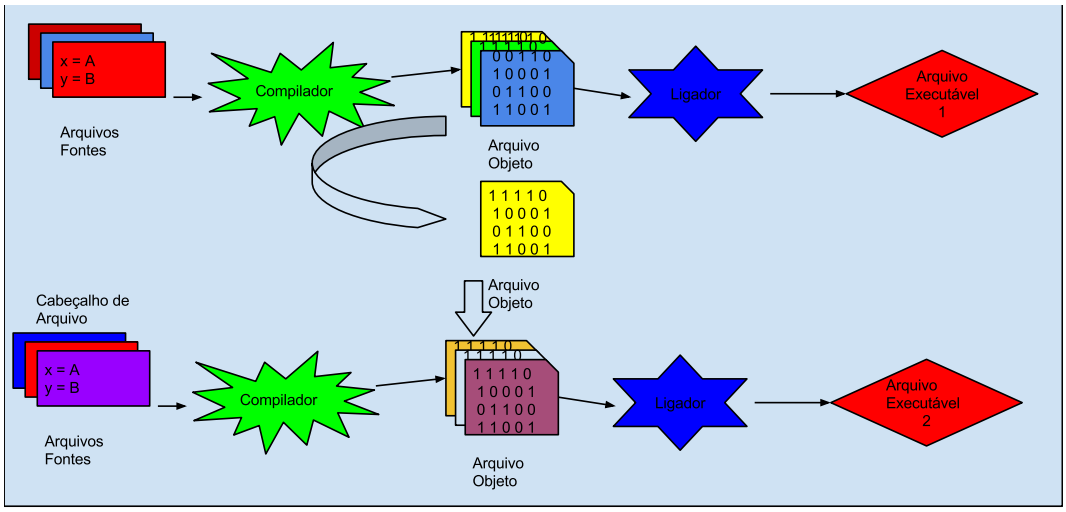

Figura 3: Reuso com biblioteca estática (STEVANOVIC, 2014, pág. 54), com adaptações.

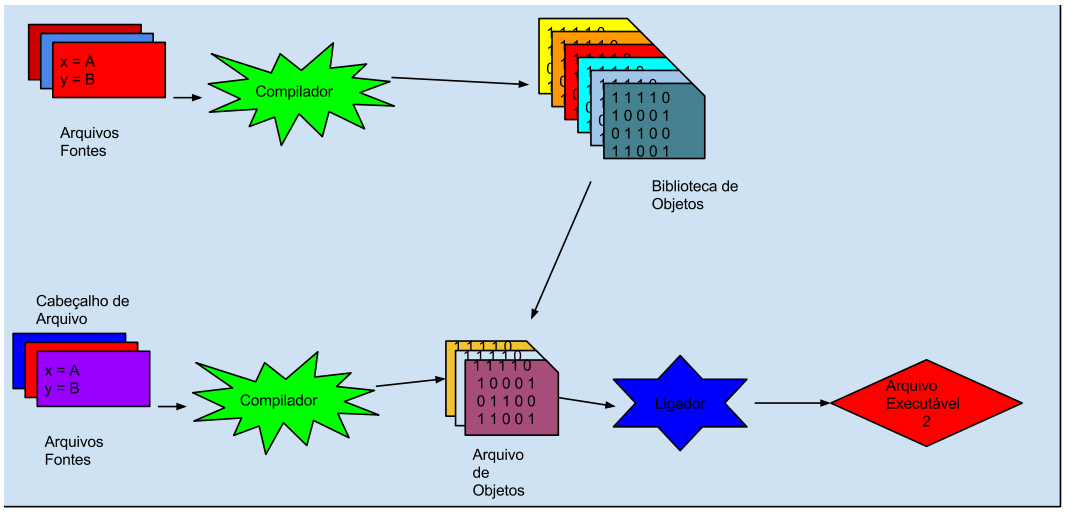

Figura 4: Reuso (maneira trivial) (STEVANOVIC, 2014, pág. 54), com adaptações.

ter informações que não são utilizadas no projeto. Bibliotecas estáticas, por estes motivos, não são utilizadas com tanta frequência nos dias de hoje.

1.2.12 Bibliotecas Dinâmicas

Ao contrário das bibliotecas estáticas, as bibliotecas dinâmicas são códigos objetos que podem ser carregados durante a execução de um projeto, como representado na Figura 5 (STEVANOVIC, 2014, pág. 57). Bibliotecas dinâmicas não aumentam o tamanho do código binário do projeto final mas, no entanto, a execução do projeto necessita da utilização de um arquivo externo (normalmente nomeado com o sufixo `.so` ou `.dll`), que contém as informações a serem carregadas (WHEELER, 2015b). Em ambiente Linux as bibliotecas devem ser registradas em uma variável de ambiente chamada `LD_LIBRARY_PATH`, que possui o caminho de todas as bibliotecas dinâmicas que podem ser utilizadas.

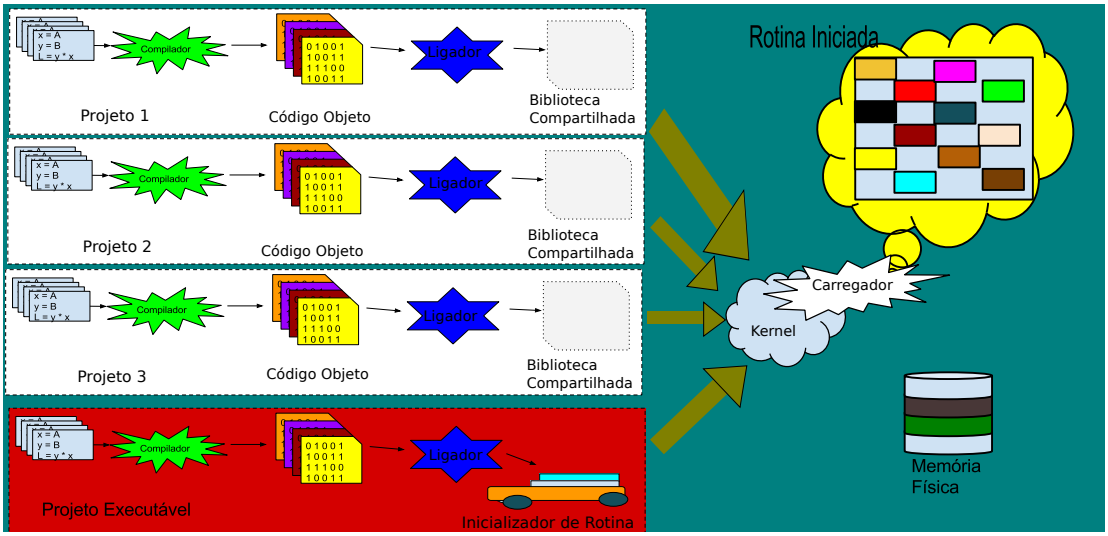

Figura 5: Biblioteca dinâmica (STEVANOVIC, 2014, pág. 55), com adaptações.

1.3 Métodos para a redução do tempo de compilação

1.3.1 Guardas de Inclusão

Em C++, a inclusão de um arquivo de código fonte em outro arquivo fonte é feita através da diretiva de pré-processamento chamada `#include`, que pode ser utilizada como mostrado no Código 1.5.

Código 1.5: Diretiva de pré-processamento para inclusão de arquivo

```
1 // biblioteca de sistema
2 #include <nome do arquivo>
3
4 ou
5
6 // outras bibliotecas
7 #include "nome do arquivo"
```

Diretivas `#include` normalmente são utilizadas várias vezes em um projeto. No entanto, o preprocessador não é capaz de verificar se um arquivo já foi adicionado, o que pode ocasionar um erro de duplicação de definição de estruturas e elementos do código. Um exemplo deste problema é mostrado na Figura 6 (STEVANOVIC, 2014, pág. 57).

Para correção deste problema foram criadas as guardas de inclusão (no inglês chamada de *include guards* ou *guard conditions*), que são diretivas de pré-processamento utilizada para verificar se um identificador já está definido ou não. Existem varias versões de guardas de inclusão, as quais serão citadas adiante (LAKOS, 1996, pág. 80).

Normalmente, arquivos-cabeçalho (ou *headers*) são utilizados para declarações de estruturas, variáveis, funções e macros. Tais arquivos são identificados pelos sufixo `.h`,

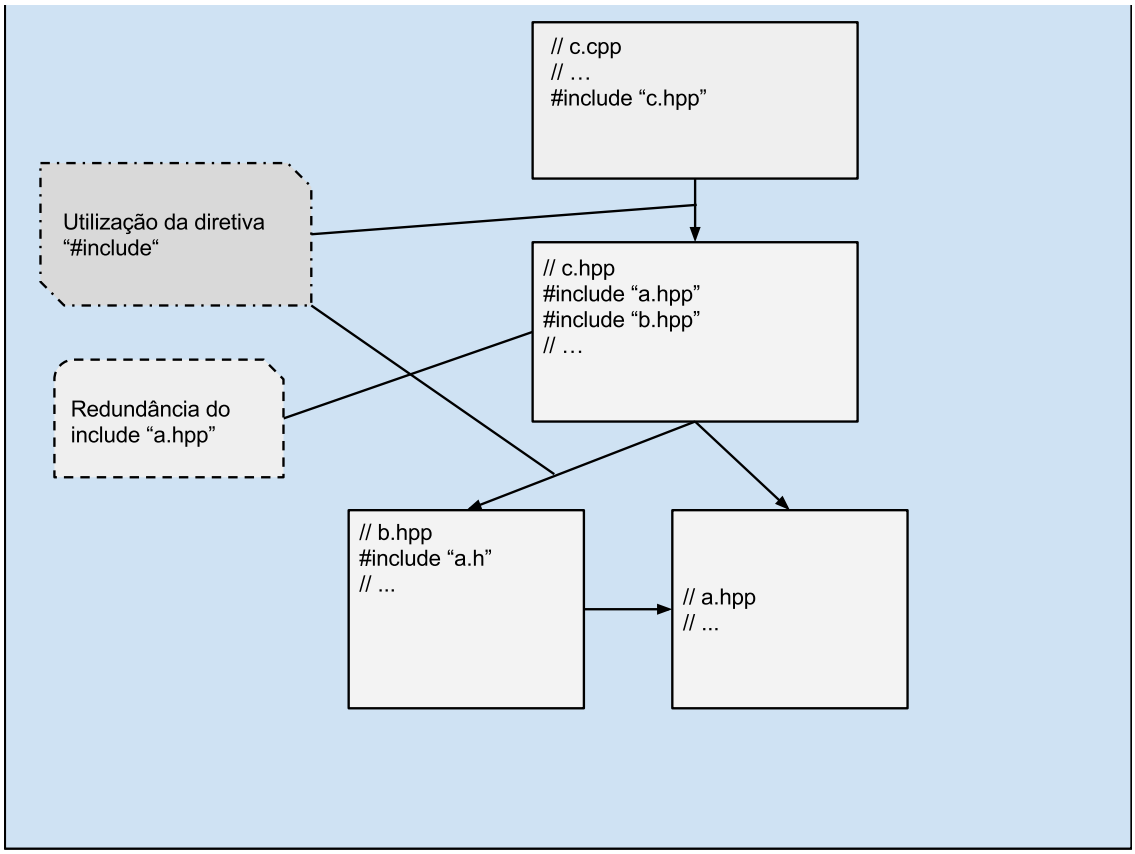

Figura 6: Includes redundante (LAKOS, 1996, pág. 80), com adaptações.

`.hpp`, `.hxx` ou `.hpp`. As implementações das funções e definições de variáveis são feitas, na maioria das vezes, em arquivos denominados fontes (ou *sources*), cujas extensões mais comuns são `.c`, `.cpp`, `.cxx`, `.cc`, `.c++`, entre outros (FOUNDATION, 2015a).

Há quatro tipos de guardas de inclusão:

1. Interna ao arquivo *header*:

   A diretiva `#ifndef` é utilizada para verificar se um identificador foi definido (forma negativa, para a forma positiva a diretiva é `#ifdef`); a diretiva `#define` é utilizada para definir um identificador; e a diretiva `#endif` e utilizada para finalizar uma condição (diretivas `#ifdef` e `#ifndef`) (LAKOS, 1996).

   Caso o identificador já esteja definido, o preprocessador irá ignorar qualquer informação que que esteja entre as diretivas `#ifndef` e `#endif`; caso não esteja definido, o identificador serão então definido e as informações que seguem a diretiva serão incluídas no arquivo resultante (LAKOS, 1996, pág. 80).

   Os Códigos 1.6, 1.7 e 1.8 ilustram uma guarda de inclusão interna ao arquivo *header*.

Código 1.6: Arquivo c.hpp contendo guardas de inclusão interna

```
1 // c.h  declaracao de variavel
```

```
2  // verficar se o simbolo INCLUDE_C esta definido
3  #ifndef INCLUDE_C
4  // define um simbolo INCLUDE_C
5  #define INCLUDE_C
6
7  #include "a.hpp"       // importa o arquivo "a.hpp"
8  #include "b.hpp"       // importa o arquivo "b.hpp"
9
10 ...                  // define as estruturas
11
12 #endif               // fim da condicional "#ifndef"
```

Código 1.7: Arquivo b.hpp que inclue o arquivo a.h

```
1  // b.hpp
2  #ifndef INCLUDE_B
3  #define INCLUDE_B
4
5   #include "a.hpp"
6
7   // declaracao de estruturas
8   ...
9  #endif
```

Código 1.8: Arquivo a.hpp contendo guardas de inclusão interna

```
1  // a.hpp    arquivo de declaracao
2  #ifndef INCLUDE_A
3  #define INCLUDE_A
4
5    // declaracao de estruturas
6    ...
7
8  #endif
```

2. Externa ao arquivo de *header*:

   Na guarda de condição externa são utilizadas as mesmas diretivas da interna, sendo a única diferença é a localização das diretivas: elas antecederão cada uso da diretiva `#include`. Um exemplo da utilização de guarda de condição externa é mostrado nos Código 1.9, 1.10 e 1.11 (LAKOS, 1996, pág. 80).

Código 1.9: Arquivo c.hpp contendo guardas de inclusão externa

```
1  // c.hpp arquivo de declaracao de estruturas de c
2
3  #ifndef INCLUDE_A
4  #define INCLUDE_A
5  /* inclue o arquivo a.hpp caso o simbolo
```

```
6     INCLUDE_A nao esteja definido */
7  #include "a.hpp"
8  #endif
9
10 #ifndef INCLUDE_B
11 #define INCLUDE_B
12  /* inclue o arquivo b.hpp caso o simbolo
13     INCLUDE_B nao esteja definido */
14  #include "b.hpp"
15 #endif
16
17  ... //implementa as declaracoes
```

Código 1.10: Arquivo a.hpp com guarda de inclusão externa

```
1  // a.hpp          declaracao das estruturas de a.hpp
2
3  ...
```

Código 1.11: Arquivo b.hpp com guarda de inclusão externa

```
1  //b.hpp         declaracao das estruturas de b.hpp
2
3  ...
```

3. Redundância:

   Consiste em utilizar, simultaneamente, as guardas de inclusão internas e externas.

   Segundo John Lakos (LAKOS, 1996, pág. 82), esta guarda de inclusão é a mais trabalhosa de ser mantida mas, no entanto, é a que mais reduz o tempo de compilação de um projeto. Ele aplicou a técnica de guardas de inclusão com e sem redundância, e obteve os resultados mostrados nas Tabelas 1 e 2.

Tabela 1: Amostra de Redundância com 10 *headers* por linha

Quantidade de Arquivos	Sem Redundância(SR)	Com Redundância(CR)	SR/CR
1	0,2 segundos	0,2 segundos	1,0 segundos
2	0,2 segundos	0,2 segundos	1,0 segundos
4	0,3 segundos	0,3 segundos	1,0 segundos
8	0,5 segundos	0,3 segundos	1,67 segundos
16	0,7 segundos	0,4 segundos	1,75 segundos
32	1,5 segundos	1,1 segundos	3,0 segundos
64	0,2 segundos	0,2 segundos	1,0 segundos
128	25,9 segundos	3,5 segundos	74,0 segundos
256	126,5 segundos	13,6 segundos	9,3 segundos
512	702,3 segundos	61,6 segundos	11,4 segundos
1024	4378,5 segundos	306, segundos6	14,28 segundos

4. *Pragma Once*

   Pragma once é uma diretiva de pré-processamento que possui o objetivo de garantir que um arquivo será lido apenas uma vez: caso uma diretiva `#include` seja

Tabela 2: Amostra de Redundância com 100 headers por linha

Quantidade de Arquivos	Sem Redundância(SR)	Com Redundância(CR)	SR/CR
1	0,2 segundos	0,2 segundos	1,0 segundos
2	0,2 segundos	0,2 segundos	1,0 segundos
4	0,4 segundos	0,3 segundos	1,33 segundos
8	0,7 segundos	0,4 segundos	1,75 segundos
16	1,7 segundos	0,5 segundos	3,4 segundos
32	5,8 segundos	0,9 segundos	6,44 segundos
64	22,1 segundos	2,0 segundos	11,05 segundos
128	89,5 segundos	5,2 segundos	17,21 segundos
256	376,5 segundos	17,1 segundos	22,02 segundos
512	1697,4 segundos	68,6 segundos	24,74 segundos
1024	8303,8 segundos	330,6 segundos	25,12 segundos

utilizada novamente e o arquivo já foi incluído anteriormente, o mesmo não será aberto. No entanto, esta implementação se tornou obsoleta na implementação no gcc (FOUNDATION, 2015b). Os Códigos 1.12, 1.13 e 1.14 exemplificam o uso desta diretiva.

Código 1.12: Arquivo c.hpp com guarda de inclusão `#pragma once`

```
1  // c.hpp
2  /* pragma once indicar que o
3     arquivo c.hpp sera
4     incluido apenas 1 vez */
5  #pragma once
6  #include "a.hpp"
7  #include "b.hpp"
8
9   // declaracoes de c.hpp
10
11  ...
```

Código 1.13: Arquivo a.hpp com guarda de inclusão `#pragma once`

```
1  // a.hpp
2  #pragma once
3
4   // declaracoes de a.hpp
5
6   ...
```

Código 1.14: Arquivo b.hpp com guarda de inclusão `#pragma once`

```
1  //b.hpp
2  #pragma once
3
4   // declaracoes de b.hpp
5
6   ...
```

1.3.2 *Forward Declaration*

Em C++, todas as entidades (variáveis, funções, classes, estruturas, uniões, etc) deve sem declaradas ou definidas antes de serem referenciadas. Definir uma classe antes dela ser utilizada não é possível quando as duas classes diferentes referenciam uma a outra, gerando uma referência cíclica (SKINNER, 1992).

Considere um exemplo de duas classes, Doutor e Paciente, apresentadas no Código 1.15. Neste exemplo, a classe Doutor requer a declaração ou definição da classe Paciente, e a classe Paciente necessita de uma declaração ou de uma definição da classe Doutor. Desta forma, a compilação geraria um erro, pois antes de utilizar uma referência é necessário declarar ou definir esta referência.

Código 1.15: Implementação de classes Paciente e Doutor

```
1  // doutor_paciente.cpp
2
3  class Doutor {
4
5    // implementacoes privadas
6    private:
7
8    /* a classe Doutor necessita saber que existe
9       a referencia para a classe Paciente */
10   Paciente* p;
11
12   ...
13
14   // implementacoes publicas
15   public:
16
17   ...
18
19 };
20
21 class Paciente {
22
23  // implementacoes privadas
24  private:
25
26  /* a classe paciente necessita saber que
27  existe a classe Doutor, no entanto
28  doutor so existe se souber da referencia
29  de paciente */
30  Doutor * d;
31
32  ...
33
```

```
34  // implementacoes publicas
35  public:
36
37  ...
38
39 };
```

Para resolver este problema é preciso utilizar uma referência incompleta (*forward*) da classe Doutor e da classe Paciente, conforme apresentado no Código 1.16.

Código 1.16: Implementação de classes utilizando *forward declaration*

```
1  // forward declaration
2  class Doutor;
3  class Paciente;
4
5  class Doutor   {
6    // implementacoes privadas
7    private:
8
9    /* a referencia incompleta de Paciente existe,
10    entao Doutor pode armazenar a
11    referencia do paciente */
12
13   Paciente* p;
14
15   ...
16
17   // implementacoes publicas
18   public:
19
20   ...
21
22 };
23
24 class Paciente {
25   // implementacoes privadas
26   private:
27
28   Doutor * d;
29
30   ...
31
32   // implementacoes publicas
33   public:
34
35   ...
36
```

```
37 };
```

A declaração incompleta (*forward declaration*) de uma classe somente podem ser utiliza em arquivos na forma de ponteiro (utilizando o operador `*`) ou referência (utilizando o operador `&`), pois estas não requerem uma definição de classe completa porque, em C++, é alocado uma quantidade fixa de memória para estes tipos de variáveis (SKINNER, 1992, pág. 111).

Um exemplo incorreto de utilização de *forward declaration* seria utilizar um construtor da classe Doutor é mostrado no Código 1.17.

Código 1.17: Uso incorreto de *forward declaration*

```
1    // forward declaration
2    class Doutor;
3
4    // utilizacao da estrutura Doutor
5    Doutor doutor();
```

Como já mencionado, qualquer entidade deve ser declarada ou definida antes de ser utilizada, então o caso acima é válido para funções, variáveis, estruturas e uniões, entre outros. A Tabela 3 mostra um exemplo de *forward declaration* para alguns destes tipos de estruturas.

Tabela 3: Exemplos de *forward declaration*

Definição	**Declaração**
`int x;`	`extern int x;`
`typedef struct Foo{int x;};`	`typedef struct Foo;`
`class Foo { int x;public: Foo() ... };`	`class Foo;`
`int Add(int x ,int y) { return x + y; }`	`int Add(int x, int y);`
`union Foo { int x; char x; } Foo;`	`union Foo;`

Na linguagem C++ há uma separação entre a implementação e a interface de classe (MEYERS, 2005, pág. 140). Na maioria dos casos, a definição de uma classe possui detalhes de sua implementação, como nos Código 1.18 e 1.19.

Código 1.18: Declaração da classe Pessoa

```
1
2  // pessoa.hpp   - declaracao da classe
3  #ifndef PESSOA_HPP
4  #define PESSOA_HPP
5
6  // conhecer detalhes de implementacao da string
7  #include <string>
8  // conhecer detalhes de implementacao da Data
```

```
9  #include "data.hpp"
10
11 class Pessoa{
12 public:
13
14   Pessoa(std::string nome,Data data);
15   std::string meu_nome() const;
16   Data meu_aniversario() const;
17
18 private:
19   // detalhes de implementacao na declaracao
20   std::string nome;
21
22   // detalhes de implementacao na declaracao
23   Data aniversario;
24 };
25
26 #endif
```

Código 1.19: Definição da classe Pessoa

```
1  // pessoa.cpp implementacao da classe
2  #include "pessoa.hpp"
3
4  Pessoa::Pessoa(std::string nome, Data data)
5          : nome(nome),aniversario(data)
6  {
7  }
8
9  std::string
10 Pessoa::meu_nome() const
11 {
12         return nome;
13 }
14 Data
15 Pessoa::meu_aniversario() const
16 {
17         return aniversario;
18 }
```

Definir uma classe com detalhes de implementação em sua definição não é uma boa prática de programação, uma vez que ela não pode ser compilada sem conhecer as definições das classes utilizadas em sua implementação para realizar alocação de memória. Isso conduz à utilização da diretriz `#include`, gerando dependência entre arquivos (MEYERS, 2005, pág. 140).

Uma maneira de contornar este problema é utilizar uma *forward declaration* e a

armazenar referências das classes, de modo que as definições das classes não possuam dependência uma das outras. Isto evita que definições de classes sejam modificadas com frequência e reduz a quantidade de recompilações devidas às dependência entre os arquivos. Veja os Códigos 1.20 e 1.21.

Código 1.20: Arquivo definição da classe Pessoa, utilizando *forward declaration*

```
1  // pessoa.hpp
2  #ifndef PESSOA_HPP
3  #define PESSOA_HPP
4
5  // header para string forward declaration
6  #include <bits/stringfwd.h>
7
8  // forward declaration
9  class Data;
10
11 class Pessoa{
12 public:
13         Pessoa(std::string& nome,Data *data);
14         std::string& meu_nome() const;
15         Data* meu_aniversario() const;
16
17 private:
18 // referencia para string
19   std::string& nome;
20
21 // referencia para Data
22   Data* aniversario;
23 };
24
25 #endif
```

Código 1.21: Definição da classe Pessoa, utilizando *forward declaration*

```
1  // pessoa.cpp -  implementacao
2  #include "pessoa.hpp"
3  #include <string>
4
5  Pessoa::Pessoa(std::string& nome, Data* data)
6         : nome(nome),aniversario(data)
7  {
8  }
9  std::string&
10 Pessoa::meu_nome() const
11 {
12         return nome;
13 }
```

```
14 Data*
15 Pessoa::meu_aniversario() const
16 {
17         return aniversario;
18 }
```

O Código 1.20 foi feito o uso da diretiva `#include <bits/stringfwd.h>`. Em C++ existem arquivos-cabeçalhos que possuem a implementação de *forward declaration*. A Tabela 4 lista alguns *forward declaration* que podem ser utilizados em C++ segundo o tipo de biblioteca padrão gcc-gnu (FOUNDATION, 2015c).

Tabela 4: Tabela de Algumas *forward declaration* padrões do C++

Biblioteca	**Alguns includes do header**
`#include <bits/stringfwd.h>`	`std::string, std::wstring, std::u16string, std::u32string;`
`#include <iosfwd>`	`std::filebuf, std::fstream, std::ifstream, std::iostream, std::istream, std::ofstream, std::ostream, std::ostringstream, std::stringbuf, std::streambuf, entre outros;`
`#include <bits/localefwd.h>`	`std:has_facet, std::isalnum, std::isalpha, std::iscntrl, std::isdigit,std::isgraph, std::islower, std::isprint, std::ispunct, std::isspace, std::issuper, entre outros;`
`#include <bits/algorithmfwd.h>`	`std::adjacent_find, std::any_of, std::binary_search, std::copy_backward, std::copy_if, std::count_if, std::count, std::find_end, std::find_if, entre outros.`

1.3.3 Makefile

Make é uma ferramenta utilizada para determinar automaticamente quais trechos de código de um grande projeto precisam serem recompilados após alguma iteração de desenvolvimento. Esta ferramenta foi implementada por Richard Stallman e Roland McGrath, desde a versão 3.76 até os dias de hoje vem sendo mantida por Paul D.Smith (WHEELER, 2015b). O manual de utilização do `makefile` é disponibilizado no portal GNU (FOUNDATION, 2004).

Para utilizar a ferramenta *make* é necessário a criação de um arquivo chamado `makefile`, que contém as descrições de relações entre os arquivos em um projeto e comandos necessários para realizar as atualizações em cada arquivo (WHEELER, 2015b). Depois de definir o arquivo `makefile`, utiliza a chamada de sistema apresentada no Código 1.22.

Código 1.22: Chamada de sistema para executar o programa make

```
1    $ make
```

O programa make realiza as devidas verificações nos códigos fontes e nos artefatos, e caso seja necessário recompilar um arquivo, o `makefile` contém as instruções necessárias para o tal procedimento (FOUNDATION, 2004, pág. 1).

1.3.4 Estrutura básica de um arquivo `makefile`

Arquivos `makefile` contém 5 tipos básicos de elementos: regras explicitas, regras implícitas, definição de variáveis, diretivas e comentários.

1. Regras Explícitas: conjunto de regras que informam quando e como refazer as construções de arquivos. Ela é definida pela estrutura apresentada no Código 1.23.

Código 1.23: Regras explicitas de Makefile

```
1
2  alvo ... : pre-requisitos ...
3      procedimentos
4    ...
5  ou
6
7  alvo ... : pre-requisitos ... ; procedimentos
8      procedimentos
9      ...
```

O **alvo** é utilizado para nomear os artefatos que devem ser gerados pelo projeto. O alvo também pode ser utilizado para fazer a chamada de execução de um trecho específico do Makefile, sendo passando como parâmetro na utilização do comando make.

Os **pré-requisitos** são os endereços de arquivos utilizados como entrada para a criação do alvo. Este campo é opcional, pois podem existir instruções que não necessitam de arquivos de entrada, como por exemplo a instrução `clean` (tipicamente utilizada para remoção de todos os códigos objetos gerados a partir dos códigos do projeto) (FOUNDATION, 2004, pág. 1).

Os **procedimentos** são as ações que devem ser executadas a partir do comando make. Este bloco pode conter mais de um comando em uma linha, ou vários comandos em linhas separadas (FOUNDATION, 2004, pág. 1).

2. Regras Implícitas: conjunto de regras que informam quando e como refazer uma classe de arquivos baseadas em seus nomes e padrões (FOUNDATION, 2004, pág. 11). Um exemplo de utilização de padrões é apresentado no Código 1.24. (FOUNDATION, 2004, pág. 19)

Código 1.24: Utilização de padrões no `makefile`

```
1
2 # gerar  objetos .o utilizando arquivos .c
3 %.o : %.c
4     procedimentos
5     ...
```

3. Definição de Variáveis: é uma linha que especifica um valor para uma variável que pode ser substituída no texto posteriormente (SKINNER, 1992, pág. 111). Veja o Código 1.25 para um exemplo de definição de variável.

Código 1.25: Definição e utilização de variável

```
1 objetos = main.o command.o display.o files.o search.o utils.o
2
3 clean:
4     rm $(objetos)
```

4. Diretivas: são instruções que indicam ao make ações especiais a medida que é realizada a leitura do arquivo Makefile (FOUNDATION, 2004, pág. 22). Estas instruções podem ser:

   - Realizar a leitura de um outro `makefile`, através da diretiva include, seguida do nome do arquivo.
   - Utilizar ou ignorar parte do `makefile` (SKINNER, 1992, pág. 22). O Código 1.26 ilustra o uso desta diretiva.

Código 1.26: Makefile com diretiva condicional

```
1
2 ibs_for_gcc = -lgnu
3 normal_libs =
4
5 foo: $(objects)
6 ifeq ($(CC),gcc)
7 $(CC) -o foo $(objects) $(libs_for_gcc)
8 else
```

```
9  $(CC) -o foo $(objects) $(normal_libs)
10 endif
```

  - **ifeq:** Diretiva que começa a condicional e especifica a condição. Ela contém 2 argumentos, separados por virgula e entre parêntesis. Caso os argumentos forem iguais as linhas entre o **ifeq** e o **else** serão executadas, caso contrário serão ignoradas (FOUNDATION, 2004, pág. 77).
  - **else:** Diretiva que marca o início das instruções a serem executadas caso a condição do **ifeq** falhe. Esta diretiva é opcional (FOUNDATION, 2004, pág. 77).
  - **endif:** Diretiva que finaliza a condição. Toda diretiva condicional condição deve ser terminada com endif (FOUNDATION, 2004, pág. 77).

- Definir variáveis com mais de uma linha. Com a utilização da diretiva *#define* e da diretiva *#endif* é possível realizar a definição de uma variável em mais de uma linha (FOUNDATION, 2004, pág. 69). O Código 1.27 ilustra esta situação.

Código 1.27: Makefile com definição de variável em múltiplas linhas

```
1 bar= "BAR"
2 define two-lines =
3 echo foo
4 echo $(bar)
5 endef
6 all:
7 $(two-lines)
```

5. Comentários: Em arquivos Makefile a utilização de caractere `#` em uma linha faz com que todos o caracteres que o sucederem sejam ignorados. Caso seja necessário a utilização deste caractere basta precedê-lo com o caractere \ (barra invertida) (FOUNDATION, 2004).

1.3.5 Padrões de um arquivo `makefile` para ferramentas GNU

Tendo como referência o manual do GNU Make[9], Seção 15.6, existem padrões que podem ser utilizados em arquivo `makefile`. Alguns dos mais comuns são:

1. **all**: instrução executada por default pelo `makefile`, realiza as verificações do arquivos de código fonte e faz atualizações nos códigos objetos, se necessário;

2. **install**: chama o make all e copia os arquivos executáveis, bibliotecas, e os demais artefatos do projeto para os locais adequados, de acordo com as necessidades do projeto;

[9] <http://www.gnu.org/software/make/manual/make.pdf>

3. **uninstall**: deleta todos os arquivos copiados pelo comando make install;
4. **clean**: deleta todos os arquivos e diretórios criados na construção do programa final;
5. **distclean**: remover os arquivos que foram gerados por algum comando make, seja na configuração, seja na construção de um programa;
6. **info**: gera um arquivo de informações sobre o programa;
7. **dist**: gera um arquivo `tar` cujo nome contém a versão e o nome do projeto e que inclui todo o conteúdo do diretório atual;
8. **check**: gera o programa e realiza testes;
9. **installcheck**: gera o programa, instala e realiza testes;
10. **installdirs**: cria os diretórios onde os arquivos devem ser instalados.

1.3.6 Executando o `make`

Depois de gerado o arquivo `makefile` utilizando as regras descritas nas seções anteriores, o comando `make` pode ser utilizado para executar as regras descritas. O comando pode ser utilizado como mostrado no Código 1.28 e conforme apresentado na Tabela 5,

Código 1.28: Utilizando o programa make

```
1        $ make [opcoes] ... [Alvo]
```

onde **alvo** é a instrução que será executada pelo *make*(A omissão deste parâmetro leva a execução do alvo *all*) e **opções** são *flags* de controle utilizadas pelo utilitário *make*. Exemplos de *flags* estão definidas na Tabela 5:

Tabela 5: Exemplos de flags que podem ser utilizadas no make

Flag	**Descrição**
`-C dir` , `-directory=dir`	Modifica o diretório antes de fazer a leitura do `makefile`
`-f file`, `-file=file`, `-makefile=FILE`	Realiza a leitura de um `makefile` específico
`-i`, `-ignore-errors`	Ignora erros na reconstrução de arquivos
`-I dir`, `-include-dir=dir`	Especifica o diretório que inclue os `makefiles`
`-j [jobs]`, `-jobs[=jobs]`	Especifica a quantidade de *threads* que podem ser executadas em paralelo. Caso não seja passado a quantidade de *threads* ele não terá limites na criação de threads.

1.3.7 *Pimpl Idiom* - Implementação privada

Ponteiro de implementação é a técnica utilizada para armazenar a implementação privada de uma estrutura em um ponteiro, que será declarado nos arquivos de *header* e implementado nos arquivos de implementação, através da utilização da declaração incompleta (*Forward Declaration*).

Esta técnica, ao ser utilizada, tem como benefício minimizar a interdependência entre estruturas e, por consequência, diminui o tempo de recompilação, além de esconder partes que não deveriam ser mostradas na declaração de uma classe, pois estas deveriam ser privadas (COPLIEN, 1992). O Código 1.29 ilustra esta técnica;

Código 1.29: Exemplo Utilização Pimpl Idiom

```
1   // book.h
2   #ifndef  BOOK_H
3   #define BOOK_H
4
5   #include <iostream>
6   using namespace std;
7
8   class Book
9   {
10  public:
11    Book(string name);
12    void read();
13  private:
14    class BookImpl;
15    BookImpl* m_impl;
16  };
17  #endif
18
19  //book.cpp
20  class Book::BookImpl
21  {
22  public:
23    BookImpl(string name,int paginas=0,read=false);
24    string get_name();
25  private:
26    string name;
27    int paginas;
28    bool read;
29  };
30
31
32
33  Book::BookImpl::BookImpl(string name,int paginas,bool read)
34    : name(name), paginas(paginas),read(read)
```

```
35      {
36      }
37
38
39      Book::BookImpl::string get_name()
40      {
41          return this->name;
42      }
43
44      Book::Book(string name)
45        : m_impl(new BookImpl(name))
46      {
47      }
48
49      void
50      Book::read()
51      {
52        cout << this->m_impl->get_name() <<endl;
53      }
54
55      //main.cpp
56      #include <iostream>
57      #include "book.h"
58      using namespace std;
59      int main()
60      {
61        Book a("eBook");
62        a.read();
63        return 0;
64      }
```

1.3.8 *Flags* de Otimização de Baixo Nível

Como visto na Seção 1.1.1 é possível otimizar o código para gerar objetos e executável menores e com tempo de execução mais rápida, no entanto ao executar este processo, o compilador perde tempo nestas otimizações de modo que o tempo de compilação aumenta. Para controlar estas otimizações, os compiladores recebem como *flags* (parâmetros) que indicam quais tipos de otimização pode ser aplicadas pelo compilador.[10]

No gcc/g++[11] [12], as flags de otimizações de compilação podem ser:

- -O0 : reduz o tempo de compilação e faz a depuração produzir os resultados esperados. Este é o padrão caso nenhuma *flag* seja passada;

[10] <https://gcc.gnu.org>
[11] <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>
[12] <https://gcc.gnu.org/onlinedocs/gnat_ugn/Optimization-Levels.html#Optimization-Levels>

- -Ofast : descarta estritas conformidades com padrões;
- -O : realiza otimização do tamanho do código e do tempo de execução do programa. No entanto não realiza otimizações que tenham grandes perdas no tempo de compilação;
- -O1 : aplica uma otimização moderada, mas não degrada significativamente o tempo de compilação;
- -O2 : realiza otimização total, gerando o código altamente otimizado e com o tempo mais lento de compilação;
- -O3 : aplica otimização total como o -O2 , no entanto liga mais flags;
- -Os : aplica as otimizações de -O2 que não implicam no aumento do tamanho do código, e também otimizações adicionais para a redução do tamanho do código;
- -Og : realiza otimizações que não interferem da depuração, oferecendo um nível de otimização razoável e mantendo a compilação rápida.

Caso seja utilizado mais de uma *flag* de otimização, a última passada que será executada e as outras serão desconsideradas.

1.3.9 Ferramentas

- **Ccache**

  Ccache[13] é uma ferramenta utilizada para armazenar compilações anteriores e guarda os resultados, detectando quando os mesmos serão necessários para reduzir o retrabalho do processo de compilação.

  Esta ferramenta tem suporte às linguagens C, C++, Objective-C and Objective-C++. Possui as vantagens : gerencia o tamanho de *cache*, armazena compilações que contém avisos (*warnings*), baixa sobrecarga, opcionalmente utiliza "*hard link*" para evitar cópias e comprime arquivos para reduzir o tamanho de uso do disco. Apesar de suas vantagens, esta ferramenta pode apenas compilar um único arquivo ou link simbólico, funciona somente com compiladores gcc/g++ ou similares e algumas *flags* de otimização não são suportadas.

  Para utilizar o *ccache* é necessário adicionar a palavra *ccache* antes de cada comando de compilação. Para remover cache de compilação anterior será utilizado ccache -c ativar o limpador de ccache e ccache -C para remover a cache completamente.

[13] <https://ccache.samba.org/download.html>

- **Gold**

  Gold é uma ferramenta de linkagem de objetos mais rápida que o linkador padrão do Linux o ld. Esta ferramenta foi desenvolvida apenas para realizar linkagem de arquivos do tipo ELF sem utilizar a biblioteca BFD (*Binary File Descriptor library*) que possui suporte a manipulação de vários formatos de arquivos binários, pelo pesquisador da Google, Ian Lance Taylor, que realizou a remoção da biblioteca e utilização de menos camadas de abstrações.

  O *linker gold* vem por padrão nos pacotes do GNU e pode ser utilizado no gcc/g++ passando a *flag* -Wl,-fuse-ld=gold[14], caso nenhuma *flag* seja passada, será utilizado o *linker* padrão *ld* que utiliza a biblioteca BFD. O *linker gold* para ser utilizado em sistemas com *back-end llvm* é necessário instalar o *plugin llvm-gold-plugin*[15].

  Para sistemas operacionais como um Mac OS e Windows esta ferramenta não é suportada, uma vez que estes sistemas possuem formato de arquivo binário diferente do ELF. Windows possui os sistemas de arquivo PE ou PE32+, enquanto o Mac OS utiliza PEF ou XCOFF[16].

[14] <https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html>
[15] <http://llvm.org/docs/GoldPlugin.html>
[16] <https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats#cite_note-apple-pef-9>

2 Metodologia

Este capítulo foi elaborado para melhorar o entendimento das atividades realizadas para a produção deste trabalho. O fluxo da Figura 7 define estas atividades.

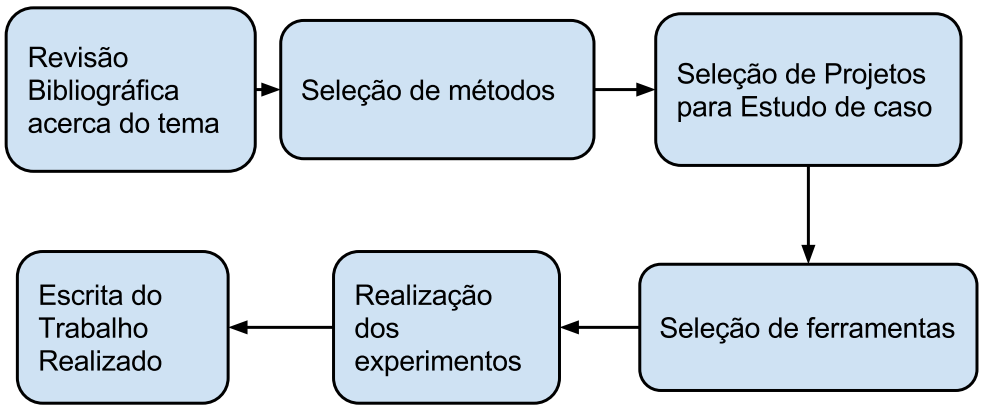

Figura 7: Fluxo de atividades para a resolução destes trabalho

2.1 Seleção de Métodos

Para seleção de métodos foi realizada uma priorização dos métodos, técnicas e ferramentas encontradas durante a revisão bibliográfica. A priorização levou em conta o esforço da implementação/execução dos métodos, o tempo para a montagem de cada caso e o recurso computacional necessário para a criação destes.

Os métodos selecionados foram:

- *Include Guards*, descrito na Seção 1.3.1;
- *Forward Declaration*, descrito na Seção 1.3.2;
- Makefile, descrito na Seção 1.3.3;
- Otimização de baixo nível descrito na Seção 1.3.8;
- *Pimpl Idiom* descrito na Seção 1.3.7;
- As ferramentas de otimização descritas na Seção 1.3.9;

2.2 Seleção de Projetos

2.2.1 Método de Seleção

Para a seleção dos projetos foram analisadas 3 plataformas de controle de informação de *software* livre: *Openhub*, *Github*, *Gitlab*.

Openhub, conhecido como Black Duck Open Hub, é uma plataforma que busca dados através dos sistemas de versionamento (tais como CVS, Git, Subversion, Bazaar e Mercurial), exibindo estatísticas sobre o histórico de um projeto e métricas do *software*. Esta plataforma não foi selecionada devido a possuir um sistema de busca baseada em *tags*, o que dificultava encontrar projetos específicos, pois o desenvolvedor que deve prover informações do projeto, e em alguns casos estas informações estão incoerentes e ou redundantes. Além disso não havia sistema de busca avançado.

Gitlab é uma plataforma de gerenciamento de *software* que permite os desenvolvedores criarem uma instância própria da ferramenta para o armazenamento de um projeto de *software* em uma infraestrutura própria. Esta ferramenta foi descartada pois também possui sistema de *tags* para localizar projetos de uma linguagem, não permite selecionar múltiplas tags e não possui filtro de pesquisa avançada.

Github é uma plataforma de gerenciamento de projetos de *software* que utiliza o sistema de versionamento *git*. Esta plataforma foi selecionada devido a possibilidade de fazer pesquisa avançada utilizando *tags*, selecionar quantidade de *forks*, linguagem principal do projeto, conteúdo de texto dentro da descrição de um projeto e quantidade de estrelas dadas para um projeto.

A pesquisa avançada foi realizada no dia 27 de agosto de 2015 e resultou na seleção dos projetos mostrados na Tabela 6 (que podem ser visualizados com maior detalhe na Seção A do Apêndice). Os criterios de seleção foram:

- **Busca por palavra chave nas informações dos projetos**

  i) **mac os** - esta palavra foi selecionada para encontrar projetos que poderiam ser compilados em um sistema operacional da Apple. Este sistema tem como padrão o compilador clang e gcc/g++ como *front-end* e como *back-end* possui o LLVM.

  ii) **windows** - esta palavra foi selecionada para encontrar que possui compatibilidade com suporte ao compiladores gcc/g++.[1]

[1] Para instalar um compilador é necessário utilizar uma ferramenta de criação de ambiente de desenvolvimento como *cygwin*, *mingw*, *Visual Studio*, entre outros.

iii) **linux** - esta palavra foi selecionada para encontrar projetos que seja possível compilar em um sistema operacional Linux. Este vem por padrão com o compilador g++/gcc.

- **Linguagem**

  i) **language: C++** - diretriz utilizada para restringir a busca de projetos que utilizam a linguagem C++ na maior parte de seu código.

- **Quantidade de Forks**

  i) **> 10 forks** - diretriz para selecionar projetos com mais de 10 *forks*.

- **Quantidade de Estrelas**

  i) **< 10000 stars** - diretriz utilizada para projetos com menos de 10 mil estrelas, indicando que pelo menos 10 mil pessoas gostaram do projeto e este é fortemente utilizado.

- **Selecionar os 5 primeiros projetos**;

Código 2.1: Busca avançada github

```
1     linux windows mac os  language:C++  stars:<10 00 forks: > 10
```

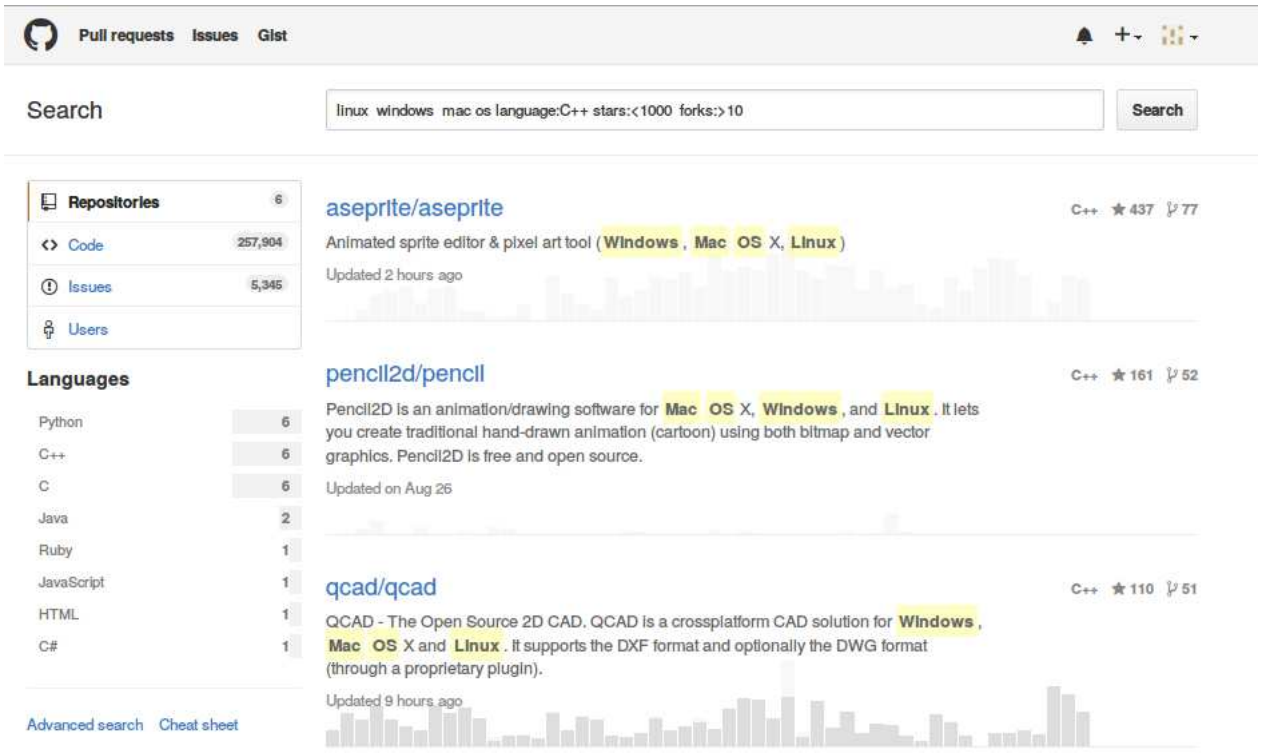

Figura 8: Pesquisa avançada Github parte 1

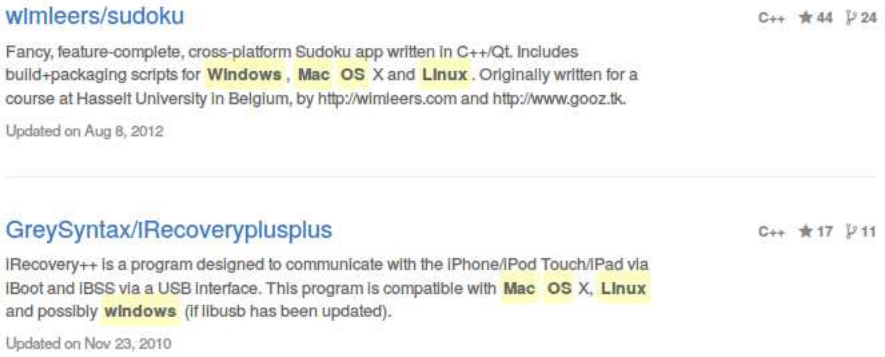

Figura 9: Pesquisa avançada Github parte 2

Tabela 6: Projetos Selecionados

Projeto	**Linguagem**	**Estrelas**	**Forks**	**Porcentagem em C++**	**Gerador de Makefile**
Aseprite	C++	437	77	62,1%	CMake
Pencil	C++	161	22	54,6%	QMake
Qcad	C++	110	51	80,8%	QMake
Sudoku	C++	44	24	94,9%	Qmake
iRecoveryplusplus	C++	17	11	84,7%	-

2.3 Coleta de Dados

Nesta seção será abordado a criação do ambiente para a coleta dos dados, detalhando configurações da máquina utilizada, bibliotecas e software instalados, ambiente vírtual criado e quais foram os impedimentos encontrados na criação do ambiente.

2.3.1 Ambiente Físico

Para realizar a coleta de dados o ambiente foi montando utilizando uma máquina com as especificações listadas na Tabela 7.

Sistema Operacional	*Ubuntu* 14.04
Kernel	3.19
Processador	Intel i5 2.0Ghz
Memória Ram	4G

Tabela 7: Configurações do Ambiente Físico

Nesta máquina foi instalado os programas:

- **VirtualBox** (4.0) : é uma ferramenta *open source* sobre a licença GPL v2 que permite a virtualização de máquinas.

- **Vagrant** - (1.7.4) : ferramenta de controle de máquina virtual possibilitando fácil acesso, criação, destruição, ligar e desligar máquinas virtuals.

- **Git** (2.5.1) : ferramenta de controle de versão descentralizada com o mesmo objetivo do *SubVersion*(**SVN**).

2.3.2 Ambiente Virtual

- **Linux**

  O ambiente virtual Linux foi criado utilizando a box ubuntu/trusty64[2], com a configuração do Vagrantfile representada no Código 2.2 e com especificações da máquina virtual mostradas na Tabela 8. Depois que construída a máquina virtual foram instalados os pacotes utilizando o gerenciador de pacotes apt, e estão listados na Tabela 9.

  Em ambiente Linux o compilador padrão de C++ que vem instalado é o gcc/g++, este foi criado pela GNU e contém um conjunto de ferramentas para realizar pré-processamento, compilação, montagem e link-edição. No *front-end* este compilador inclui as linguagens C, C++, Objective-C, Fortran, Java, Ada e Go, bem como bibliotecas (stdlibc++, libgcj, entre outras).

Código 2.2: Vagrantfile com configurações da máquina virtual linux

```
1  # Vagrantfile
2  Vagrant.configure(2) do |config|
3      config.vm.define 'linux' do |linux|
4          linux.vm.box = "ubuntu/trusty64"
5          linux.vm.network 'private_network', ip: "10.10.10.4"
6          linux.vm.provider "virtualbox" do |vm|
7              vm.cpus = 2
8              vm.memory = 2048
9          end
10     end
11 end
```

Tabela 8: Configurações do Ambiente Virtual Linux

Sistema Operacional	*Ubuntu* 14.04 x86_64
Kernel	3.13.0-45-generic
Processador	2 processadores Virtuais
Memória Ram	2GB

[2] <https://atlas.hashicorp.com/ubuntu/boxes/trusty64>

Tabela 9: Pacotes instalados no Ambiente Virtual Linux

Pacote	**Versão**
Qt5-default	5.5.1
Qt4-default	4.8.7
libusb-1.0-0/libusb-1.0-0-dev	1.0.17-1ubuntu2
libreadline6/libreadline6-dev	6.3-4ubuntu2 amd6
zlib1g/zlib1g-dev	1.2.8.dfsg-1ubuntu1 amd6
zlibc	0.9k-4.1
g++/gcc	4:4.8.2-1ubuntu6
git	1.9.1-1
cmake	2.8.12.2
ccache	3.1.9

- **Mac OS Yosemite**

  Para o ambiente Mac OS foi utilizada uma box com o sistema operacional na versão Yosemite 10.10.4[3], com as configurações do Vagrantfile representada no Código 2.3 e com especificações da máquina virtual mostradas na Tabela 10. Após a máquina criada foi necessário instalar os pacotes utilizando o gerenciador de pacotes brew[4], e foram listados os pacotes listados na tabela 11. Além dos listados foi necessário instalar o pacote "**MacOSX10.4.Universal.pkg**"[5] disponibilizado pela Apple[6].

  Em um ambiente Mac OS o LLVM[7] é o *back-end* do compilador e como *front-end* padrão possui o clang/clang++, que foi projetado para ser 3 vezes mais rápido que o gcc/g++. As linguagens aceitas pelo clang são Objective-C, Fortran, Ada, Haskell, Java, Python, Ruby, ActionScript, GLSL, Julia, entre outras.

Código 2.3: Vagrantfile com configurações da máquina virtual Mac OS Yosemite

```
1  # Vagrantfile
2  Vagrant.configure(2) do |config|
3    config.vm.define 'mac_os_x' do |mac_os_x|
4      mac_os_x.vm.box = "mac_os_x"
5      mac_os_x.vm.network 'private_network', ip: "10.10.10.3"
6      mac_os_x.vm.synced_folder "projetos", "/vagrant", type: "
         rsync"
7
```

[3] <http://files.dryga.com/boxes/osx-yosemite-0.2.1.box>
[4] <http://brew.sh/>
[5] <https://github.com/macartur-tcc/aseprite/blob/master/INSTALL.md>
[6] <https://developer.apple.com/downloads/>
[7] LLVM: infraestrutura de compilação utilizando o principio de modularização e reuso, propondo otimizar a compilação, execução e link-edição de programas escritos em linguagens variadas

```
8           mac_os_x.vm.provider "virtualbox" do |vm|
9             vm.gui = true
10            vm.memory = 2024
11          end
12        end
13      end
```

Tabela 10: Configurações do Ambiente Virtual Mac OS Yosemite

Sistema Operacional	*OS X Yosemite* x86_64
Kernel	Darwin Kernel 14.4.0
Processador	2 processadores Virtuais
Memória Ram	2GB

Tabela 11: Pacotes instalados no Ambiente Virtual Mac OS Yosemite

Pacote	**Versão**
qt5	5.5.1
qt4	4.8.7
libusb	1.0.20
readline	6.3-8
zlib	1.2.8
git	1.9.1-1
cmake	3.3.0
binutils	2.25-4
ccache	3.2.3

- **Windows**

  Para o ambiente virtual Windows foi utilizado uma box com o sistema operacional Windows7[8], com as configurações do Vagrantfile mostradas no Código 2.4, e com especificações da máquina virtual mostradas na Tabela 12. Para a utilização de um ambiente de desenvolvimento no windows é necessário a instalação do software *cygwin*[9], que permite a instalação o dos pacotes necessários para a criação do ambiente igual ao Unix. Utilizando o *cygwin* é necessário instalar todos os pacotes para o sistema conseguir compilar programas em C/C++, para isto foram instalados os programas que estão listadas na tabela 13.

Código 2.4: Vagrantfile com configurações da máquina virtual Windows 7

[8] <https://vagrantcloud.com/datacastle/boxes/windows7>
[9] <https://www.cygwin.com/>

```
#Vagrantfile
Vagrant.configure(2) do |config|
  config.vm.define 'win7' do |win|
    win.vm.box = 'win7ie8'
    win.vm.network 'private_network', ip: '10.10.10.5'
    win.vm.provider 'virtualbox' do |vm|
          vm.cpus = 2
          vm.gui = true
          vm.memory = 1024
    end
  end
end
```

Tabela 12: Configurações do Ambiente Virtual Windows 7

Sistema Operacional	*Windows 7* x86
Kernel	CYGWIN_NT-6.1
Processador	2 processadores Virtuais
Memória Ram	2GB

Tabela 13: Pacotes instalados no Ambiente Virtual Windows 7

Pacote	**Versão**
qt5-devel-tools qt5-translations todos os pacotes que começam com este nome libQt5	5.5.1-1
qt4-designer-plugin-webkit qt4-devel-tools qt4-qtconfig	4.8.7-1
libusb1.0	1.0.19-1
libreadline-devel	6.3.8-1
zlib	1.2.8
git	2.5.3-1
ccache	3.1.9-2
cmake	3.3.2-1
cygwin-devel	2.2.1
cygwin64-gcc-core	4.9.2-1
gcc-core	
gcc/g++ make	4.9.3-1 4.1-1

2.3.3 Impedimentos durante a coleta

Durante a montagem do ambiente foram encontrados alguns impedimentos que impactaram em alguns dados a serem coletados. Estes impedimentos foram:

i) Ambiente com *mingw* ou *cygwin* não possui suporte a biblioteca glibc[10], o que fez com que o projeto Qcad não pode ser compilado neste ambiente pois este necessita do header exec_info.h que é encontrado na biblioteca glibc.

ii) Durante a montagem do ambiente no Windows 7 o projeto Aseprite teve que ser ser adicionado o *header* `<cstdlib>`, pois neste ambiente não foi encontrado este header no projeto. Estas modificações foram colocadas no Github[11] e não impediu a compilação do projeto.

2.4 Exemplos de Uso

Esta seção apresenta como foi elaborado a execução de cada método e ferramenta na realização da coleta de dados. Primeiro foi utilizado um benchmark com guardas de inclusão executado em cada um dos diferentes ambientes e em seguida a aplicação dos métodos e ferramentas utilizando os projetos selecionados.

2.4.1 Guardas de Inclusão

Os casos de guardas de inclusão foram realizados utilizando *script* de *benchmark*, na qual os métodos detalhados na Seção 1.3.1 foram aplicados a 10 mil arquivos com uma pequena quantidade de Código 2.5 e incluidos 3 vezes ao arquivo **main.cpp** utilizando 2.6. Cada um destes foram aplicados aos diferentes ambientes de desenvolvimento.

Código 2.5: Template de arquivo .hpp utilizado no *benchmark*

```
1
2   // <NUMERO>.hpp
3
4   const int int<NUMERO> = <NUMERO>;
```

Código 2.6: Template de arquivo main.cpp utilizado no *benchmark*

```
1
2     // main.cpp
3
4     /* headers a serem incluidos   */
5
6     ...
```

[10] <https://www.gnu.org/software/gnulib/manual/html_node/execinfo_002eh.html>
[11] <https://github.com/macartur-tcc/>

```
7
8     int main(){return 0;}
```

Os scripts abordam os métodos:

- guarda de Inclusão Externa;
- guarda de Inclusão Interna;
- *pragma once*;
- guarda de Inclusão Interna primeiro que *pragma once*;
- *pragma once* primeiro que Guarda de Inclusão Interna;
- guarda de Inclusão Externa + *pragma once*;
- redundância de Guarda de Inclusão.

2.4.2 Aplicação de métodos e uso de ferramentas externas

Para a realizar a coleta foi necessário identificar quais as técnicas que poderiam ser aplicadas em cada um dos projetos. As tabelas 14, 15 e 16 mostram esta analise.

Tabela 14: Técnicas que podem ser aplicadas ao código

Projeto	**pragma once**	**Forward Declaration**	**Pimpl Idiom**
iRecoveryplusplus	SIM	SIM	SIM
Sudoku	SIM	SIM	SIM
Pencil	SIM	NÃO	SIM
Aseprite	SIM	NÃO	NÃO
Qcad	SIM	NÃO	NÃO

Tabela 15: *Flags* que podem ser utilizadas

Projeto	**Makafile**	**Otimização de Baixo Nível**
iRecoveryplusplus	SIM	SIM
Sudoku	SIM	SIM
Pencil	SIM	SIM
Aseprite	SIM	SIM
Qcad	SIM	SIM

Tabela 16: Ferramentas que podem ser utilizadas

Projeto	**Ccache**	**Gold**
iRecoveryplusplus	SIM	SIM
Sudoku	SIM	SIM
Pencil	SIM	SIM
Aseprite	SIM	SIM
Qcad	SIM	SIM

2.4.2.1 Alteração de Código

Para as alterações de código, cada caso foi criado uma branch com suas modificações e estas estão no repositório destinado ao TCC[12].

1. *Pragma Once*

   Neste estudo de caso os arquivos de header foram alterados com a inclusão da diretiva "#pragma once" no inicio de cada arquivo. No projeto aseprite está sendo removida a diretiva, pois todos os arquivos já possuem. Estas alterações estão na branch "*pragma_once*" dos projetos que foi possível aplicar.

2. *Forward Declaration*

   Neste estudo de caso os *headers* foram alterados para reduzir a quantidade de *headers* de todos os cabeçalhos e das implementação. Para isso foram removidos *headers* redundantes e que referenciava ponteiros e referências de endereço. Estas alterações estão na branch "*forward_declaration*"dos projetos que foram possíveis aplicar.

3. *Técnicas de Pimpl Idiom*

   Neste estudo de caso atributos e métodos privados foram removidos do cabeçalho e incluídos nos arquivos de implementação, como no exemplo de Código 1.29 da Seção 1.3.7 que demostra como aplicar a técnica. Estas alterações estão na branch "*pimpl*"dos projetos que foi possível aplicar.

2.4.2.2 Parametrização de Build

Antes da realização da compilação será feito 2 tipos de parametrização, utilizando os métodos detalhados nas Seções 1.3.3 e 1.3.8. Sendo assim será utilizado:

1. *makefile*

[12] <https://github.com/macartur-tcc/>

No momento da utilização do makefile será utilizado as *flags*: -j2,-j4,-j6,-j8 e -j10. Com esta abordagem será investigada a relação entre a compilação e a quantidade de *jobs* (*threads*).

2. Otimização de Baixo Nível

   Todos os arquivos Makefile serão alterados para receber a *flags*: `-O`, `-O0`, `-O2`, `-Ofast`, `-O3`, `-Os`,`-Og`. Com estes resultados será possível verificar a relação do tempo gasto com o tempo de otimização do compilador.

2.4.2.3 Ferramentas Externas

Para a realização da coleta de dados utilizando ferramentas externas será utilizado as ferramentas detalhadas na Seção 1.3.9 e seguindo os comandos:

1. *ccache*

   Neste estudo de caso em cada projeto será adicionado a palavra ccache antes de cada linha de compilação. Será utilizado as configurações padrão do ccache, e antes de cada compilação para ter certeza que toda a cache será removida utilizará o comando ccache -C e ccache -c.

2. *linker gold*

   Neste estudo de caso será ativado o ld.gold utilizando a *flag*:-Wl,-fuse-ld=gold como mostrado na seção 1.3.9. Assim a compilação irá deixar de utilizar o linker padrão e utilizará o linker otimizado para sistema ELF.

2.4.2.4 Passos de Compilação dos projetos

- iRecoveryplusplus

  Para compilar o projeto iRecoveryplusplus é necessário realizar o clone do projeto no github, e compilar utilizando os comandos no Código 2.7.

Código 2.7: Clonado e compilando o projeto iRecoveryplusplus

```
1
2     # clonando o projeto
3     $ git clone https://github.com/macartur-tcc/iRecoveryplusplus.
         git
4     $ cd iRecoveryplusplus
5
6     # compilando no Linux, Mac OS Yosemite ou Windows 7
7     $ make linux
8
9     # compilar utilizando o linker gold no Linux
```

```
10      $ make linux  CC="gcc -Wl,-fuse-ld=gold" CXX="g++ -Wl,-fuse-ld=
            gold"
11
12      # compilar utilizando a ferramenta ccache
13      $ make linux CC="ccache gcc" CXX="ccache g++"
14
15      # removendo arquivos binarios gerados
16      rm bin/*
```

- **Sudoku**

  Para compilar o projeto Sudoku é necessário realizar o clone do projeto no github utilizando os comandos no Cógido 2.8.[13]

Código 2.8: Clonado e compilando o Projeto Sudoku

```
1
2       # clonando projeto do github
3       $ git clone https://github.com/macartur-tcc/sudoku.git
4       $ cd sudoku/src
5
6       # gerando makefile
7       $ qmake Sudoku_release.pro
8
9       # compilando no Linux, Mac OS ou Windows 7
10      $ make
11
12      # compilar no Linux utiizando o linker gold
13      $ make  CC="gcc -Wl,-fuse-ld=gold" CXX="g++ -Wl,-fuse-ld=gold"
14
15      # compilando utilizando a ferramenta ccache
16      $ make CC="ccache gcc" CXX="ccache g++"
17
18      # remover arquivos binarios gerados
19      $ make clean
```

- **Pencil**

  Para compilar o projeto Pencil é necessário realizar o clone do projeto no github e executar os comandos mostrados no Código 2.9

Código 2.9: Clonado Projeto Pencil

```
1       #clonando o projeto do github
2       $ git clone https://github.com/macartur-tcc/pencil.git
3       $ cd pencil
4
```

[13] Neste projeto e necessário remover a *flag* -archi386 do arquivo *Makefile* gerado na raiz, pois em sistemas Mac OS esta *flag* causa erro.

```
5     # gerando makefile para ambiente Linux e Windows 7
6     $ qmake -r
7
8     # gerando makefile para ambiente Mac OS Yosemite
9     $ qmake -r RESOURCES=./pencil.qrc
10
11    # compilando
12    $ make
13
14    # compilando utilizando a ferramenta ccache
15    $ make CC="ccache gcc" CXX="ccache g++"
16
17    # compilando utilizando gold
18    make CC="gcc -Wl,-fuse-ld=gold" CXX="g++ -Wl,-fuse-ld=gold"
19
20    # removendo arquivos binarios gerados
21    $ make clean
```

- **Aseprite**

  Para compilar o projeto Aseprite é necessário fazer o clone do projeto, gerar os arquivos makefile seguindo os comandos no Código 2.10.

Código 2.10: Clonado Projeto Aseprite e criando diretório de Compilação

```
1     # clonado projeto do github
2     $ git clone https://github.com/macartur-tcc/aseprite.git
3     $ cd aseprite
4
5     # criando diretorio para compilar o projeto
6     $ mkdir build
7     $ cd build
8
9     # gerando makefile do projeto no Linux e Windows 7
10    $ cmake -G .. "Unix Makefiles"
11
12    # gerando makefile no Mac OS Yosemite
13    $ cmake -G ... "Unix Makefiles" -DCMAKE_OSX_ARCHITECTURES:
         STRING=i386  -DCMAKE_OSX_DEPLOYMENT_TARGET:STRING=10.4  -
         DCMAKE_OSX_SYSROOT:STRING=/SDKs/MacOSX10.4u.sdk
14
15    # compilando o projeto com ccache no Linux Windows 7
16    $ cmake -G .. "Unix Makefiles"  -DCMAKE_CXX_COMPILER="ccache"
         -DCMAKE_CXX_COMPILER_ARG1="g++" -DCMAKE_C_COMPILER="ccache"
         -DCMAKE_C_COMPILER_ARG1="gcc"
17
18    # compilando projeto com ccache no Mac OS Yosemite
19    $ cmake -G .. "Unix Makefiles"  -DCMAKE_CXX_COMPILER="ccache"
         -DCMAKE_CXX_COMPILER_ARG1="g++" -DCMAKE_C_COMPILER="ccache"
```

```
        -DCMAKE_C_COMPILER_ARG1="gcc"  -DCMAKE_OSX_ARCHITECTURES:
        STRING=i386  -DCMAKE_OSX_DEPLOYMENT_TARGET:STRING=10.4  -
        DCMAKE_OSX_SYSROOT:STRING=/SDKs/MacOSX10.4u.sdk

    # compilando com o linker gold no Linux
    $ CC="gcc  -Wl,-fuse-ld=gold" CXX="g++ -Wl,-fuse-ld=gold" cmake
        .. -G "Unix Makefiles"

    # removendo arquivos binarios
    $ make clean
```

- **Qcad**

  Para compilar o projeto Qcad é necessário clonar o projeto no github, gerar os arquivos makefile e compilar utilizando os comandos descritos no Código 2.11.

Código 2.11: Clonado Projeto Qcad

```

    #clonando projeto do github
    $ git clone https://github.com/macartur-tcc/qcad.git
    $ cd qcad

    # compilando no Linux e Windows 7
    $ qmake -r

    # compilando no Mac OS Yosemite
    qmake -spec macx-g++ -r

    # compilando no Linux, Windows7 e Mac OS Yosemite
    $ make

    # compilando com a ferramenta ccache
    $ make CC="ccache gcc" CXX="ccache g++"

    # compilando com o linker gold
    $ make  CC="gcc -Wl,-fuse-ld=gold" CXX="g++ -Wl,-fuse-ld=gold"

    # removendo arquivos binarios gerados
    $ make clean
```

2.5 Montagem de Scripts

Os scripts foram elaborados utilizando a linguagem de programação python, e estão sendo armazenados no github[14].

[14] <https://github.com/macartur-tcc/tcc_scripts>

2.5.1 Benchmark com Guardas de Inclusão

Para o estudo aplicado ao ambiente foi criado um script de benchmark utilizando a linguagem python, e foi detalhado no Apêndice B.

A Tabela 17 representa o modelo utilizado na coleta dos dados, contendo Guarda de Inclusão Externa (GIE), Guarda de Inclusão Interna (GII), Pragma Once (PO), Guarda de Inclusão Interna Primeiro que Pragma Once (GIIPPO), Pragma Once Primeiro que Guarda de Inclusão Interna (POPGII), Guarda de Inclusão Externa e Pragma Once (GIEPPO) e Redundância de Guarda de Inclusão (RGI).

Tabela 17: Template de amostra de Guardas de Inclusão

Tipo	Utilização de Guarda de Inclusão						
Medida	Tempos em segundos						
Amostras	**GIE**	**GII**	**PO**	**GIIPPO**	**POPGII**	**GIEPPO**	**RGI**
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
Média:							

2.6 Métodos e ferramentas aplicados a um projeto

Para realizar as compilações dos projetos foi criado um script de recompilação que segue no Código C.1, com este é possível alterar entre branch para compilar as alterações de código dos projetos, executar comandos antes e depois de cada compilação. Para receber todos estes comandos foi necessário a leitura de um arquivo de configuração de cada projeto contendo a localização do projeto, localização do makefile, comandos que serão executados antes e depois da compilação. Este arquivo foi escrito em yaml e seguindo o template do Código C.2. Para a contagem do tempo de compilação foi utilizado o módulo datetime do python, realizando a diferença do tempo inicial e final do tempo de compilação.

Após a execução do script e a coleta dos dados as Tabelas 18, 19, 20 e 21 foram preenchidas e resultou nas tabelas encontradas na Seção D.2 do Apêndice.

Tabela 18: Modelo que aplica Alteração de Código

<Nome do Projeto>	**master**	**pragma once**	**forward declaration**	**private implementation**
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
Média				

Tabela 19: Modelo aplica parametrização com *flags* de otimização

<Nome do Projeto>	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
Média							

Tabela 20: Modelo que aplica flgas de processamento paralelo

<Nome do Projeto>	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
Média					

Tabela 21: Modelo com utilização de ferramentas externas

<Nome do Projeto>	**gold**	**ccache**
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
Média		

3 Resultados

Este capítulo apresenta os resultado obtido com a aplicação dos exemplos de uso descritos no Capítulo 2.4.

Para melhor organização do trabalho, os resultados serão apresentados em seções, que correspondem à métodos correlacionados.

3.1 Guardas de Inclusão

As tabelas que contém o detalhamento das execuções dos experimentos se encontram no Apêndice D.1. A Tabela 22 sintetiza as médias destes resultados, e a Figura 15 apresenta os mesmos dados, em forma visual. Os tempos são dados em segundos, e os métodos aplicados foram abreviados com as seguintes siglas:

GIE: Guarda de Inclusão Externa

GII: Guarda de Inclusão Interna

PO: Pragma Once

GIIPPO: Guarda de Inclusão Interna Primeiro que Pragma Once

POPGII: Pragma Once Primeiro que Guarda de Inclusão Interna

GIEPPO: Guarda de Inclusão Externa e Pragma Once

RGI: Redundancia de Guarda de Inclusão

Tabela 22: Resultados das Guardas de Inclusão

	Tempo médio de compilação (em segundos)						
Sistema Operacional	**GIE**	**GII**	**PO**	**GIIPPO**	**POPGII**	**GIEPO**	**RGI**
Linux	0,2851	0,2851	1,2127	1,3179	1,2311	1,2477	0,1611
Mac OS Yosemite	0,8482	0,8149	1,0031	1,0139	0,9511	0,9097	0,5668
Windows 7	2,4015	2,3954	4,2161	4,2611	4,3152	4,2621	2,1471

De acordo com dados apresentados, pode-se observe que, para os experimentos em questão, todos os métodos que utilizam a diretiva `#pragma once` atigiram tempo de compilação superior em relação aos métodos a utilizam, independente do sistema operacional em questão. Dentre os métodos que não utilizaram esta diretiva, o método **RGI** foi o que resultou em menor tempo médio de compilação.

Em relação aos sistemas operacionais listados, o Mac OS Yosemite apresentou a menor variação dentre os diferentes métodos, apresentado comportamento aproximadamente linear (com média 0,8725 e desvio padrão de 0,1424, o que corresponde a 16% de

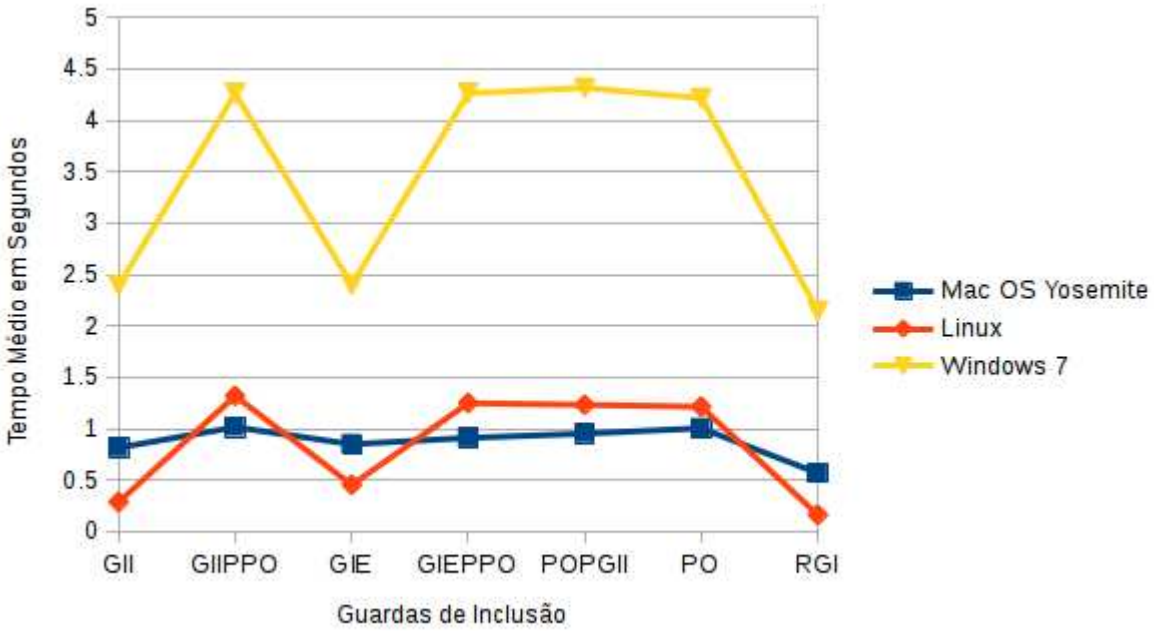

Figura 10: Dados coletados dos scripts de Guardas de Inclusão

variação em torna da média, contra 61% e 28% obtido no Linux e Windows, respectivamente). No Windows 7, o tempo de compilação dos métodos foi, na maior parte dos métodos, de 3 a 4 vezes maior que nos outros sistemas operacionais.

3.2 Métodos que envolvem a edição do código fonte

O primeiro método avaliado, que evolve a edição do código fonte, foi a guarda de inclusão `#pragma once` (apresentada na Seção 1.3.1). Este método foi aplicado em todos os 5 projetos selecionado, e os tempos médios de compilação são apresentados na Tabela 23.

Tabela 23: Resultados do `#pragma once`

Projeto	Tempo médio de compilação (em segundos)								
	Linux			Mac OS Yosemite			Windows 7		
	Sem	Com	Redução (%)	Sem	Com	Redução (%)	Sem	Com	Redução (%)
Aseprite	2362	2357	0,021	769	772	-	2441	2317	5,07
iRecoveryplusplus	1,16	0,86	25,9	2,85	2,92	-	4,88	5,02	-
Pencil	455	459	-	353	346	1,98	530	529	0,18
Sudoku	22	22	-	38	38	-	32	29	9,37
Qcad	3923	3841	2,09	3729	3290	11,77	-	-	-

Para os projetos selecionados, o impacto desta técnica não se mostrou relevante, pois em várias configurações não houve sequer ganho em relação ao tempo médio e, nos casos onde houve ganho, a porcentagem foi muito baixa (exceto no caso do iRecoveryplusplus no Linux, com 25% de ganho, mas o tempo de compilação do projeto já era muito baixo, o que afeta diretamente na proporção ganho).

O segundo método ,que envolve a edição de código fonte, foi a técnica de declaração implicita de estrutura (encontrada na Seção 1.3.2). Este método foi aplicado apenas em 2 projetos, pois os outros projetos já possuiam esta técnica. A Tabela 24 apresenta os tempos médios de compilação produzidos durante a coleta de dados.

Tabela 24: Resultados da declaração implicita de estrutura

Projeto	Tempo médio de compilação (em segundos)								
	Linux			Mac OS Yosemite			Windows 7		
	Sem	Com	Redução (%)	Sem	Com	Redução (%)	Sem	Com	Redução (%)
iRecoveryplusplus	1,17	0,9	23,7	2,85	2,7	5,26	4,97	4,3	13,48
Sudoku	22,21	20,63	7,11	38.42	36.26	5,62	32,72	27,97	14,51

Para os projetos avaliados o impacto desta técnica reduziu o tempo de compilação, no entanto não se mostrou tão relevante, uma vez que o tempo de redução máximo atingido foi de 23,7% (no qual este projeto possui um tempo de compilação muito baixo e com uma variação muito alta levando a uma perda de proporção) no Linux e as outras ocorrencias reduziram menos de 15% do tempo de compilação.

O terceiro método avaliado, que envolve a edição de código fonte, foi a implementação do ponteiro de declaração privada apresentado no Seção 1.3.7. Aplicando esta técnica a Tabela 25 foi elaborada com os tempos médios de compilação dos 3 projetos que foram alterados utilizando a técnica.

Tabela 25: Resultados da aplicação ponteiro de implementação privada

Projeto	Tempo médio de compilação (em segundos)								
	Linux			Mac OS Yosemite			Windows 7		
	Sem	Com	Redução (%)	Sem	Com	Redução (%)	Sem	Com	Redução (%)
iRecoveryplusplus	1,17	0,91	22,22	2,85	2,70	5,26	4,97	4,83	2,81
Pencil	455	454	0,22	353	353	-	530	520	1,88
Sudoku	22,21	21,39	3,69	38,42	37,82	1,56	32,72	28,78	12,04

Para os projetos avaliados esta técnica impactou na redução do tempo de compilação, no entanto entanto em projetos com tempo de compilação grande como no *Pencil*, que demorou cerca de 7 a 8 minutos para compilar, a redução foi de 0,22%, mostrando-se que esta técnica apesar de reduzir a compilação não possui um impacto relevante na compilação de um projeto.

Por fim, os dados detalhados de todos os experimentos com métodos que envolvem a edição do código fonte estão no Apêndice D.2.1.

3.3 Métodos que envolvem a parametrização do *build*

O primeiro método avaliado na parametrização de *build* é o método que utiliza flags de otimização, apresentado na Seção 1.3.8. Este método foi aplicado em todos os 5 projetos selecionados. A Tabela 26 apresenta os resultados coletados no ambiente linux.

Vale ressaltar que em todos os projetos a o gerador de makefile coloca por padrão a *flag* `-O2` no makefile.

Tabela 26: Resultados com *flags* de otimização no Linux

	Tempo médio de compilação (em segundos)							
Projeto	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**	**Melhor Redução (%)**
Aseprite	2274	2073	2357	2412	2246	**705**	2253	70
iRecoveryplusplus	1,04	**0,89**	1,17	1,12	0,90	1,13	1,03	23,93
Pencil	456	454	455	455	**453**	455	455	0,44
Sudoku	22,23	22,51	22,21	21,92	**19,65**	21,92	22,08	11,53
Qcad	3919	3925	3923	3919	**3886**	3922	3937	0,94

No ambiente linux esta técnica teve um ganho maior no projeto *Aseprite* com a utilização da *flag* `-Ofast` com uma redução de 70% do tempo de compilação. No entanto vale lembra que este projeto utiliza o gerador de makefile cmake, e os outros projetos em sua maioria utiliza o qmake. De maneira geral a maioria dos projetos tiveram redução utilizando a *flag* `-Os`.

Com o ambiente Mac OS Yosemite a Tabela 27 apresenta os resultados do tempo de compilação médio obtido utilizando esta técnica. Neste ambiente foi possível utilizar utilizar o método nos 5 projetos, no entanto a *flag* `-Og` não foi utilizada devido ao *back-end* do LLVM não suportar.

Tabela 27: Resultados com flags de otimização no Mac OS Yosemite

	Tempo médio de compilação (em segundos)						
Projeto	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**Melhor Redução (%)**
Aseprite	775	592	769	795	708	**209**	72,82
iRecoveryplusplus	2,90	**2,57**	2,85	2,83	2,76	2,86	9,82
Pencil	353	**352**	353	353	364	356	0,39
Sudoku	38,36	38,50	38,42	**38,32**	39,38	38,34	0,26
Qcad	3795	**3079**	3729	3737	3793	3786	17,43

Para este ambiente o projeto *Aseprite* também obteve o maior ganho utilizando o *flag* `-OFast`. Vale resaltar que como dito anteriormente este projeto foi criado utilizando o gerador de makefile cmake, diferente dos outros projetos. Avaliando o impacto em todos os projetos a *flag* `-O0` obteve um maior redução.

No ambiente Windows 7 a Tabela 28 apresenta o tempo de compilação médio utilizando apenas 4 projetos selecioados, pois dito anteriormente na Seção 2.3.3 o projeto A.3 não pode ser instalado devido a falta de dependências neste ambiente.

Tabela 28: Resultados com flags de otimização no Windows 7

	Tempo médio de compilação (em segundos)							
Projeto	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**	**Melhor Redução (%)**
Aseprite	2254	2092	2441	2550	2231	**620**	2184	74,6
iRecoveryplusplus	4,58	4,65	4,97	5,17	**4,53**	4,95	4,85	8,85
Pencil	516	**501**	530	534	522	539	513	5,47
Sudoku	31,15	**29,29**	32,72	35,28	31,41	35,69	30,11	16,98

Analisando os resultados dos projetos avaliados, o projeto Aseprite foi o obteve o maior ganho utilizando a *flags* `-Ofast`. No entanto de uma maneira geral a *flag* `-O0`

obteve o maior ganho na maioria dos projetos, exceto no projeto iRecoveryplusplus que possui o menor tempo de compilação dentre todos os projetos analisados.

Após a parametrização utilizando *flags* de otimização foi realizada a parametrização utilizando *flags* de processamento paralelo com a ferramenta make. Esta técnica foi mensionada na Seção 1.3.3 e foi possível utilizar todos os projetos selecionados. A Tabela 29 apresenta os dados coletados utilizando este método no ambiente Linux, e demostra o tempo médio de compilação dos projetos utilizando cada uma das *flags*.

Tabela 29: Resultados com flags de processamento paralelo no Linux

	Tempo médio de compilação (em segundos)						
Projeto	**sem flag**	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**	**Melhor Redução (%)**
Aseprite	2357	1390	**1229**	1479	1695	1439	47,85
iRecoveryplusplus	1,17	**1,11**	1,12	1,13	1,13	1,12	5,13
Pencil	455	271	**242**	257	284	288	46,81
Sudoku	22	13	13	13	14	**10**	54,54
Qcad	3923	2437	**2301**	2323	1603	1557	41,34

No ambiente Linux ao utilizar qualquer uma das *flags* o tempo de compilação é reduzido em mais de 40% em todos os projetos. De maneira geral o uso da *flag* `-j 10` não necessáriamente é o que impacta na maior redução do tempo de compilação, pois nos casos apresentados a maior parte dos projetos tiveram o menor tempo de compilação utilizando a *flag* `-j 4`.

Para o ambiente Mac OS Yosemite a Tabela 30 apresenta os resultados obtidos utilizando esta técnica. Todos os projetos selecionados foram utilizados.

Tabela 30: Resultados com flags de processamento paralelo no Mac OS Yosemite

	Tempo médio de compilação (em segundos)						
Projeto	**sem flag**	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**	**Melhor Redução (%)**
Aseprite	769	576	575	**288**	578	571	62,54
iRecoveryplusplus	2,85	2,50	2,73	2,50	2,41	**2,33**	18,24
Pencil	353	279	275	282	276	**265**	24,92
Sudoku	38	28,27	28,17	27,97	28,31	**27,94**	26,47
Qcad	3729	3596	3685	3745	3422	**3103**	16,78

No ambiente Mac OS Yosemite esta tecnica apresentou resultados diferentes do ambiente Linux.A *flag* `-j 10` resultou no menor tempo de compilação na maioria dos projetos.No entanto a redução neste ambiente não passou de 27%, exceto no projeto *Aseprite* que utilizando a *flag* obteve uma redução maior que 60%.

Com a utilização desta tecnica no ambiente Windows 7 a Tabela 31 apresenta os resultados obtidos. Apenas o projeto Qcad não foi possível ser utilizado neste ambiente, devido a necessidade de uma dependência de ambiente como mostrado na Seção 2.3.3.

Com os projetos selecionados, a maioria dos projetos apresentou uma redução superior a 50% utilizando a *flag* `-j 6`, exceto no iRecoveryplusplus em que seu tempo de compilação é muito baixo e sendo assim a redução foi pouco relevante.

Tabela 31: Resultados com flags de processamento paralelo no Windows 7

Projeto	Tempo médio de compilação (em segundos) sem flag	-j 2	-j 4	-j 6	-j 8	-j10	Melhor Redução (%)
Aseprite	2441	1140	1093	**1005**	1083	1022	58,82
iRecoveryplusplus	4,97	4,99	5,02	5,11	5,01	**4,94**	0,60
Pencil	530	262,35	259,95	**258,90**	259,15	266,47	51,15
Sudoku	32,72	17,32	17,84	**16,57**	18,43	18,14	49,36

Por fim as Tabelas apresentadas na Seção 2.6 foram preenchidas e estão disponíveis no Apêndice localizados na Seção D.2.2.

3.4 Métodos que envolvem o uso de ferramentas externas

Com a utilização de ferramentas externas o *linker gold* foi utilizado apenas no ambiente Linux, como exposto na Seção 1.3.9. A Tabela 32 apresenta os tempos médios da compilação dos 5 projetos selecionados.

Tabela 32: Resultados com uso do *linker gold*

Projeto	Tempo médio de compilação (em segundos) Linux		
	Sem	Com	Redução (%)
Aseprite	2357	2270	3,69
iRecoveryplusplus	1,17	1,00	14,52
Pencil	455	457,37	-
Sudoku	22	21,17	3,77
Qcad	3923	3847	1,94

Utilizando esta ferramenta na maioria dos projetos o tempo de compilação é considerado irrelevante pois reduzido em menos de 4% nos casos em que houve redução, exceto no projeto iRecoveryplusplus que possui o tempo de compilação muito baixo e então os ganhos são apresentados de maneira desproporcional.

A segunda ferramenta *ccache*, que é detalhada na Seção 1.3.9, foi utilizada em todos os 5 projetos selecionais. A Tabela 33 apresenta os dados coletados tendo como médida o tempo de compilação médio em segundos.

Tabela 33: Resultados com uso da ferramenta *ccache*

Projeto	Tempo médio de compilação (em segundos) Linux			Mac OS Yosemite			Windows 7		
	Sem	Com	Redução (%)	Sem	Com	Redução (%)	Sem	Com	Redução (%)
Aseprite	2357,87	540,58	77,07	769,16	209,82	72,72	2441,38	761,73	68,80
iRecoveryplusplus	1,17	1,14	2,56	2,85	2,63	7,71	4,97	5,04	-
Pencil	455	173	61,98	353	104	70,54	530	181	65,85
Sudoku	22	4	81,81	38	7	81,58	32	8	75
Qcad	3923	1823	53,53	3729	664	82,19	-	-	-

O uso da ferramenta ccache proprocionou uma redução significativa na maioria dos projetos, e com maior impacto em projetos grandes. O projeto iRecoverypluplus possui os menores tempos de redução devido ao mesmo possuir o menor tempo médio de compilação.

Os gráficos 22, 23 e 24 apresentados no Apêndice proporcionam uma melhor visualização destes dados.

Por fim os dados para uma visualização de dados mais detalhados da utilização destas ferramentas é apresentado no Apêndice D.2.3.

Parte III

Conclusão

4 Conclusão

Neste trabalho foi apresentado 5 métodos de redução do tempo de compilação: guardas de inclusão, declaração implicita de estrutura, ponteiro privado de implementação, uso de flags de compilação para otimização de baíxo nível, flags de processamento paralelo, e o uso de 2 ferramentas *linker gold* e o *ccache*.

Dentre as técnicas utilizadas as tecnicas que envolvem alteração de código não apresentaram valores satisfatórios na coleta de dados realizada. Ao contrario destes os métodos que apresentam parametrização e ferramentas obtiveram um resultado bastante satisfatório, na qual é possível reduzir mais de 50% do tempo de compilação de um projeto.

O método que apresentou melhores resultados utilizou as *flags* de processamento paralelo, na qual proporcionou uma redução minima de 45% do tempo de compilação do projeto. Dentre as ferramentas utilizadas a ferramenta *ccache* é a mais eficiente pois reduziu mais de 50% do tempo de compilação de todos os projetos apresentados.

Para a coleta de dados foi utilizado 3 ambientes diferentes, na qual utilizando virtualização foram instalados o sistema operacional Windows 7, Mac OS Yosemite e Ubuntu. Dentre estes ambientes o Windows 7, que utilizou o *cygwin* como ambiente de desenvolvimento, foi o ambiente que resultou no maior tempo de compilação dos projetos sem a utilização de nenhuma técnica ou ferramenta. O ambiente Mac OS Yosemite foi o ambiente rápido dentre os selecionados pois este utiliza o compilador com *back-end* LLVM.

Tendo estas informações este trabalho conseguiu atingir os seu objetivo geral, que foi conseguir selecionar métodos e ferramentas que consegui-se atingir uma redução em mais de 30% do tempo de compilação de projetos de código aberto selecionados.

Referências

AHO, A. V. *Compilers: Principles, Techniques and Tools*. Reading, Massachusetts, EUA: Addison-Wesley, 1986. Citado na página 33.

AHO, A. V.; ULLMAN, J. D. *The Theory of Parsing, Translation and Compiling, Vol. 1: Parsing*. Englewood Cliffs, Nova Jersey, EUA: Prentice Hall, 1972. Citado na página 33.

AHO, A. V.; ULLMAN, J. D. *Principles of Compiler Design*. Reading,Massachusetts, EUA: Addison-Wesley, 1977. Citado 2 vezes nas páginas 25 e 35.

AHO, A. V. M. S. L. . R. S. D. U. *Compiladores: Princípios, técnicas e ferramentas*. São Paulo: Addison-Wesley, Pearson, 2008. Citado 5 vezes nas páginas 29, 30, 32, 33 e 35.

ALBLAS, H.; NYMEYER, A. *Practice and Principles of Compiler Building with C*. London: Prentice Hall, 1996. Citado na página 34.

CARDOSO, B.; AULER, R. *Getting started with LLVM Core Libraries*. Birminghan Munbai: Packt Publishing, 2014. Citado na página 31.

COOPER, T. *Engineering a Compiler*. San Francisco: Morgan Kaufmann, 2003. Citado na página 29.

COPLIEN, J. O. *Advanced C++ Programming Styles and Idioms*. [S.l.]: Addison Wesley, 1992. Citado na página 55.

CRESPO, R. G. *Processadores de Linguagens: da Concepção à Implementação*. Lisboa, Portugal: IST Press, 1998. Citado na página 34.

DELAMARO, M. *Como Construir um Compilador Utilizando Ferramentas Java*. São Paulo: Novatec, 2004. Citado na página 34.

DERSHEM, H. L.; JIPPING, M. J. *Programming Languages: Structures and models*. Boston: PWS Publishing Company, 1995. Citado na página 29.

FISCHER, A. E.; GRODZINSKY, F. *The Anatomy of Programming Languages*. Englewood Cliffs, New Jersey: Prentice Hall, 1993. Citado 4 vezes nas páginas 29, 30, 31 e 32.

FISCHER, C. N.; LEBLANC, J. Crafting a compiler with c. In: . Redwood City, California: Benjamin Cummings Publishing, 1991. v. 1. Citado na página 33.

FOUNDATION, F. S. *GNU MAKE*. 51 Franklin St. Boston, 2004. Citado 4 vezes nas páginas 50, 51, 52 e 53.

FOUNDATION, F. S. *GNU GCC - Online Documentation*. 2015. Acessado: 14-06-2015. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-4.4.1/gcc/Overall-Options.html>. Citado na página 41.

FOUNDATION, F. S. *GNU GCC - Online Documentation*. 2015. Acessado: 14-06-2015. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-2.95.3/cpp_1.html>. Citado na página 44.

FOUNDATION, F. S. *GNU GCC - Online Documentation*. 2015. Acessado: 14-06-2015. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a00932.html>. Citado na página 50.

LAKOS, J. *Large-Scale C++ Software Design*. Reading,Massachusetts, EUA: Addison-Wesley Longman, 1996. Citado 5 vezes nas páginas 9, 40, 41, 42 e 43.

LEVINE, J. R. *Linkers & Loaders*. San Francisco: Morgan Kaufmann Publishers, 2000. Citado 3 vezes nas páginas 9, 37 e 38.

LEWIS, P.; ROSENKRANTZ, D. *Compiler Design Theory*. Reading, Massachusetts: Addison-Wesley, 1978. Citado na página 34.

MEYERS, S. *Effetive C++,55 Specific Ways to Improve Your Programs and Designs*. Reading,Massachusetts, EUA: Addison-Wesley, 2005. Citado 2 vezes nas páginas 47 e 48.

MUCHNICK, S. S. *Advanced Compiler Design Implementation*. San Francisco, California: Morgan Kaufmann Publishers, 1977. Citado na página 35.

PRICE, A. M. A.; TOSCANO, S. S. *Implementação de Linguagens de Programação: Compiladores: Série de Livros Didáticos Número 9*. Porto Alegre: Sagra Luzzatto, 2000. Citado na página 33.

PYSTER, A. B. *Compiler Design and Construction: Tools and Techniques*. Nova York, EUA: Van Nostrand Reinhold Company, 1988. Citado na página 34.

SCOTT, M. L. *Programming Language Pragmatics*. San Francisco: Morgam Kaufmann/Academic Press, 2009. ISBN 13:978-0-12-374514-9. Citado 11 vezes nas páginas 9, 25, 29, 30, 31, 32, 33, 34, 35, 36 e 37.

SEBESTA, R. *Conceitos de Linguagens de Programação*. Porto Alegre: Bookman, 2010. Citado na página 31.

SKINNER, M. T. *The advanced C/C++ Book*. [S.l.]: Solicon Press, 1992. Citado 3 vezes nas páginas 45, 47 e 52.

SORENSON, J.-P. T. andPaul G. *The Theory and Practice of Compiler Writing*. Nova York: McGraw-Hill, 1989. Citado na página 35.

STEVANOVIC, M. *Advanced C and C++ Compiling*. [S.l.]: Apress, 2014. 54 p. Citado 4 vezes nas páginas 9, 38, 39 e 40.

STROUSTRUP, B. *The C++ Programming Language, Fourth Edition*. [S.l.]: Addison Wesley, 2013. Citado na página 26.

WATSON, D. *High-Level Languages and Their Compilers*. Wokingham, Reino Unido: Addison-Wesley, 1989. Citado na página 34.

WHEELER, D. A. *COMO FAZER Bibliotecas de Programa*. 2015. Acessado: 11-06-2015. Disponível em: <http://www.las.ic.unicamp.br/~felipe/plh/Program-Library-HOWTO>. Citado na página 38.

WHEELER, D. A. *COMO FAZER Bibliotecas de Programa*. 2015. Acessado: 13-06-2015. Disponível em: <http://www.las.ic.unicamp.br/~felipe/plh/Program-Library-HOWTO>. Citado 3 vezes nas páginas 39, 50 e 51.

Apêndices

APÊNDICE A – Projetos Selecionados

A.1 Aseprite

Asepriter[1] é uma ferramenta para criação de *sprites*, compostas por camadas e quadros, com suporte a RGBA, paleta de 256 cores e escala de cinza. Permite visualizar a animação em tempo real e contém ferramentas para preencher contornos e polígonos. Esta ferramenta e open source e livre, e pode ser encontrada no *Github* [2] ou no site da ferramenta [3].

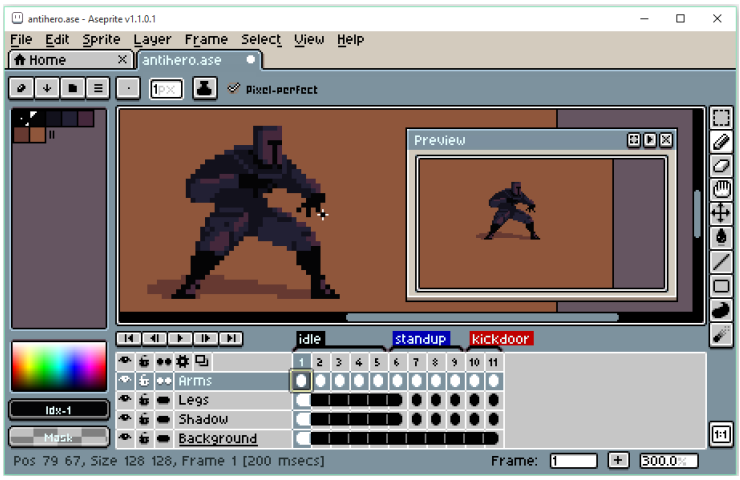

Figura 11: Imagem da ferramenta Aseprite

A.2 Pencil

Pencil[4] é uma ferramenta para desenho e animação desenhada a mão, e utilizando desenhos em formato bitmap ou gráfico vetoriais. Esta ferramenta é gratuita e open source, podendo ser encontrada em no *Github*[5] ou no site da ferramenta[6].

1 <http://www.aseprite.org/>
2 <https://github.com/aseprite/aseprite>
3 <http://www.aseprite.org/>
4 <http://www.pencil2d.org/pencil2d/>
5 <https://github.com/pencil2d/pencil>
6 <http://www.pencil2d.org>

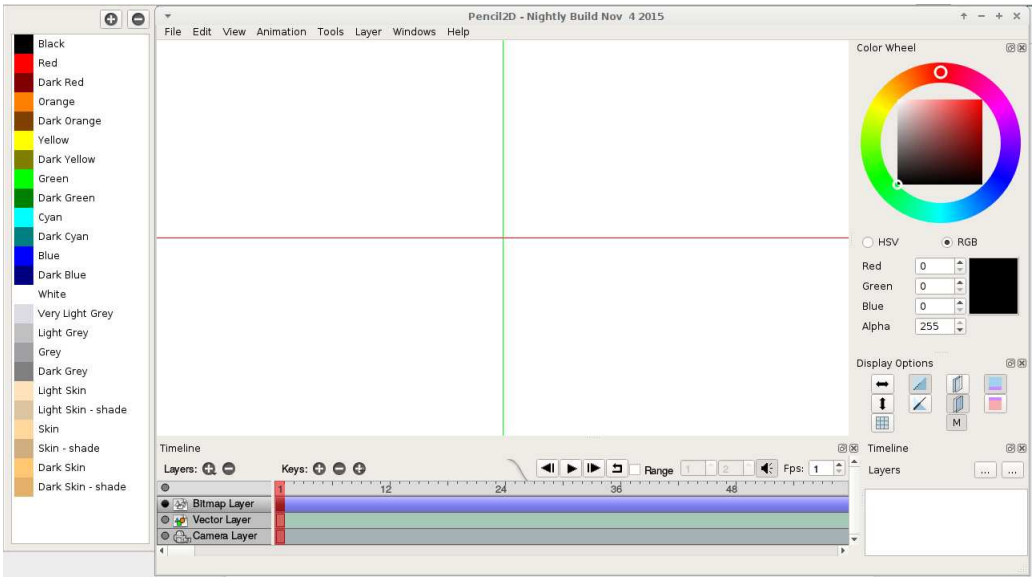

Figura 12: Imagem da ferramenta

A.3 Qcad

Qcad[7] é um software para desenho assistido por computador em duas dimensões, permitindo a criação de desenhos técnicos como planos para edifícios, interiores, peças mecânicas ou esquemas e diagramas. Esta ferramenta e open source e gratuita e podendo ser encontrada no *Github*[8] ou no site da ferramenta.[9]

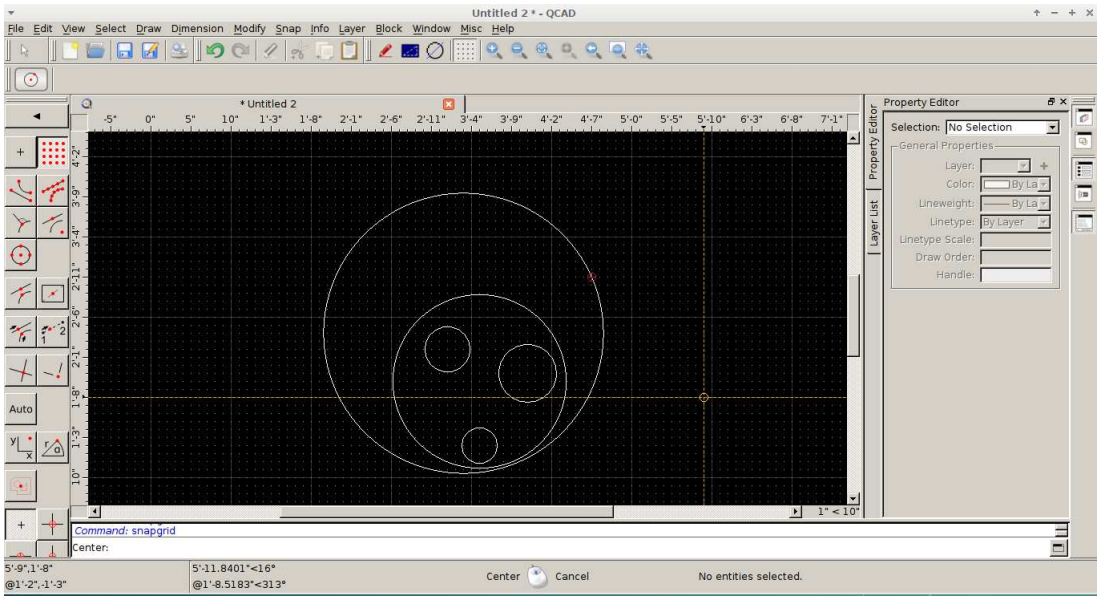

Figura 13: Imagem da ferramenta Qcad

7 <http://www.qcad.org/en/>
8 <https://github.com/qcad/qcad>
9 <http://www.qcad.org/en/>

A.4 Sudoku

Sudoku[10] é um software que foi desenvolvido em um curso na universidade de Hasselt na Bélgica, esta foi escrita em c++ e utilizando Qt. Este é um software livre e liberado para domínio publico e pode ser encontrado apenas no *Github*[11].

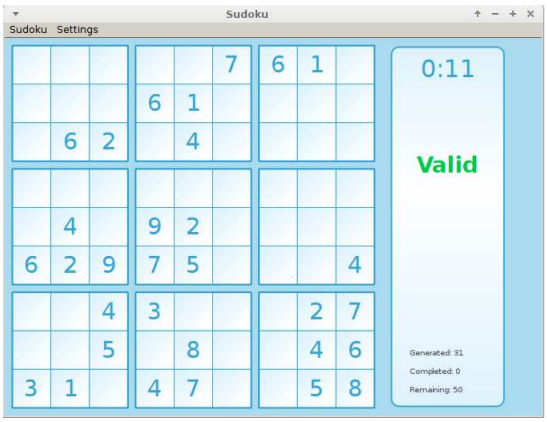

Figura 14: Imagem da ferramenta Sudoku

A.5 iRecoveryplusplsu

iRecoveryplusplus[12] é um programa projetado para realizar comunicação de iPod, iPhone e iPad iBoot/iBSS com interface USB. Este é um software livre e sobre a licença GPV v3 e pode ser encontrado no *github*[13] com suporte aos sistemas operacionais Windows, Mac OS e Linux.

10 <https://github.com/macartur-tcc/sudoku>
11 <https://github.com/macartur-tcc/sudoku>
12 <https://github.com/GreySyntax/iRecoveryplusplus>
13 <https://github.com/GreySyntax/iRecoveryplusplus>

APÊNDICE B – Script Guardas de Inclusão

B.1 Guardas de Inclusão Externa mais Pragma Once

Código B.1: Script Guardas de Inclusão Externa mais pragma once

```
1
2  # external_pragma.py
3  from os import mkdir
4  from os import path
5  from os import system
6
7  number_of_files = 10**4
8  number_of_includes = 3
9
10 folder =  "./external_pragma/"
11
12 include_directory = folder+"include"
13 include_path = folder+"include/{0}.hpp"
14 path_main_file = folder+"main.cpp"
15
16 content_of_include = """#pragma once\nconst int int{0} = {0};"""
17 end_of_main_file = "int main() {\n}\n"
18 header = """#ifndef H{0}_HPP
19 #define H{0}_HPP
20 #include "{0}.hpp"
21 #endif
22 """
23
24 def verify_directory(path_name):
25     if not path.exists(path_name):
26         mkdir(path_name)
27
28 def create_include_file(path,content):
29     f =  open(path,"w+")
30     f.write (content)
31     f.close()
32
33 def create_includes():
34     #create directory
35     # create all files
36     for number in range(0,number_of_files):
37         path = include_path.format(str(number))
38         content = content_of_include.format(str(number))
39         create_include_file(path,content)
```

```
40
41 def create_main_file():
42     #open main.cpp
43     main = open(path_main_file,"w+")
44
45     #write includes 3 times
46     for number in range(0,number_of_files):
47         for x in range(0,number_of_includes):
48             content = header.format(str(number))
49             main.write(content)
50         main.write("\n")
51
52     #write end of file
53     main.write(end_of_main_file)
54
55     #close main.cpp
56     main.close()
57
58 def copy_util_files():
59     command = "cp util/* "
60     command += folder
61     system(command)
62
63 def main():
64     verify_directory(folder)
65     verify_directory(include_directory)
66     create_includes()
67     create_main_file()
68     copy_util_files()
69
70 if __name__ == "__main__":
71     main()
```

B.1.1 Guardas de Inclusão Externa

Código B.2: Script Guardas de Inclusão Externa

```
1
2
3 # external.py
4 from os import mkdir
5 from os import path
6 from os import system
7
8 number_of_files = 10**4
9 number_of_includes = 3
10
```

```
11 folder =  "./external/"
12
13 include_directory = folder+"include"
14 include_path = folder+"include/{0}.hpp"
15 path_main_file = folder+"main.cpp"
16
17 content_of_include = """const int int{0} = {0};"""
18 end_of_main_file = "int main() {\n}\n"
19 header = """#ifndef H{0}_HPP
20 #define H{0}_HPP
21 #include "{0}.hpp"
22 #endif
23 """
24
25 def verify_directory(path_name):
26     if not path.exists(path_name):
27         mkdir(path_name)
28
29 def create_include_file(path,content):
30     f =  open(path,"w+")
31     f.write (content)
32     f.close()
33
34 def create_includes():
35     #create directory
36     # create all files
37     for number in range(0,number_of_files):
38         path = include_path.format(str(number))
39         content = content_of_include.format(str(number))
40         create_include_file(path,content)
41
42 def create_main_file():
43     #open main.cpp
44     main = open(path_main_file,"w+")
45
46     #write includes 3 times
47     for number in range(0,number_of_files):
48         for x in range(0,number_of_includes):
49             content = header.format(str(number))
50             main.write(content)
51         main.write("\n")
52
53     #write end of file
54     main.write(end_of_main_file)
55
56     #close main.cpp
57     main.close()
```

```
58
59 def copy_util_files():
60     command = "cp util/* "
61     command += folder
62     system(command)
63
64
65 def main():
66     verify_directory(folder)
67     verify_directory(include_directory)
68     create_includes()
69     create_main_file()
70     copy_util_files()
71
72 if __name__ == "__main__":
73     main()
```

B.1.2 Guardas de Inclusão Interna

Código B.3: Script Guardas de Inclusão Interna

```
1
2 # guard_only.py
3 from os import mkdir
4 from os import path
5 from os import system
6
7 number_of_files = 10**4
8 number_of_includes = 3
9
10 folder =  "./guard-only/"
11
12 include_directory = folder+"include"
13 include_path = folder+"include/{0}.hpp"
14 path_main_file = folder+"main.cpp"
15
16 content_of_include = """#ifndef H{0}_HPP
17 #define H{0}_HPP
18 const int int{0} = {0};
19 #endif
20 """
21
22 end_of_main_file = "int main() {\n}\n"
23 header = "#include \"{0}.h\"\n"
24
25 def verify_directory(path_name):
26     if not path.exists(path_name):
```

```
27          mkdir(path_name)
28
29  def create_include_file(path,content):
30      f =  open(path,"w+")
31      f.write (content)
32      f.close()
33
34  def create_includes():
35      #create directory
36      # create all files
37      for number in range(0,number_of_files):
38          path = include_path.format(str(number))
39          content = content_of_include.format(str(number))
40          create_include_file(path,content)
41
42  def create_main_file():
43      #open main.cpp
44      main = open(path_main_file,"w+")
45
46      #write includes 3 times
47      for number in range(0,number_of_files):
48          for x in range(0,number_of_includes):
49              content = header.format(str(number))
50              main.write(content)
51          main.write("\n")
52
53      #write end of file
54      main.write(end_of_main_file)
55
56      #close main.cpp
57      main.close()
58
59  def copy_util_files():
60      command = "cp util/* "
61      command += folder
62      system(command)
63
64  def main():
65      verify_directory(folder)
66      verify_directory(include_directory)
67      create_includes()
68      create_main_file()
69      copy_util_files()
70
71  if __name__ == "__main__":
72      main()
```

B.1.3 Guardas de Inclusão Interna primeiro que o Pragma Once

Código B.4: Script Guardas de Inclusão Interna primeiro que *Pragma Once*

```
1
2  # guards_pragma.py
3  from os import mkdir
4  from os import path
5  from os import system
6
7  number_of_files = 10**4
8  number_of_includes = 3
9
10 folder =  "./guards-pragma/"
11
12 include_directory = folder+"include"
13 include_path = folder+"include/{0}.hpp"
14 path_main_file = folder+"main.cpp"
15
16 content_of_include = """#ifndef H{0}_HPP
17 #define H{0}_HPP
18 #pragma once
19 const int int{0} = {0};
20 #endif
21 """
22
23 end_of_main_file = "int main() {\n}"
24 header = """#include "{0}.hpp"\n"""
25
26 def verify_directory(path_name):
27     if not path.exists(path_name):
28         mkdir(path_name)
29
30 def create_include_file(path,content):
31     f =  open(path,"w+")
32     f.write (content)
33     f.close()
34
35 def create_includes():
36     #create directory
37     # create all files
38     for number in range(0,number_of_files):
39         path = include_path.format(str(number))
40         content = content_of_include.format(str(number))
41         create_include_file(path,content)
42
43 def create_main_file():
44     #open main.cpp
```

```
45      main = open(path_main_file,"w+")
46
47      #write includes 3 times
48      for number in range(0,number_of_files):
49          for x in range(0,number_of_includes):
50              content = header.format(str(number))
51              main.write(content)
52
53      #write end of file
54      main.write(end_of_main_file)
55
56      #close main.cpp
57      main.close()
58  def copy_util_files():
59      command = "cp util/* "
60      command += folder
61      system(command)
62
63
64  def main():
65      verify_directory(folder)
66      verify_directory(include_directory)
67      create_includes()
68      create_main_file()
69      copy_util_files()
70
71  if __name__ == "__main__":
72      main()
```

B.1.4 Guardas de Inclusão com Pragma Once primeiro que Inclusão Interna

Código B.5: Script *Pragma Once* primeiro que Guardas de Inclusão Interna

```
1
2  # pragma_guards.py
3  from os import mkdir
4  from os import path
5  from os import system
6
7  number_of_files = 10**4
8  number_of_includes = 3
9
10 folder =  "./pragma-guards/"
11
12 include_directory = folder+"include"
13 include_path = folder+"include/{0}.hpp"
14 path_main_file = folder+"main.cpp"
```

```
15
16 content_of_include = """#pragma once
17 #ifndef H{0}_HPP
18 #define H{0}_HPP
19 const int int{0} = {0};
20 #endif
21 """
22
23 end_of_main_file = "int main() {\n}"
24 header = """#include "{0}.h"\n"""
25
26 def verify_directory(path_name):
27     if not path.exists(path_name):
28         mkdir(path_name)
29
30 def create_include_file(path,content):
31     f =  open(path,"w+")
32     f.write (content)
33     f.close()
34
35 def create_includes():
36     #create directory
37     # create all files
38     for number in range(0,number_of_files):
39         path = include_path.format(str(number))
40         content = content_of_include.format(str(number))
41         create_include_file(path,content)
42
43 def create_main_file():
44     #open main.cpp
45     main = open(path_main_file,"w+")
46
47     #write includes 3 times
48     for number in range(0,number_of_files):
49         for x in range(0,number_of_includes):
50             content = header.format(str(number))
51             main.write(content)
52
53     #write end of file
54     main.write(end_of_main_file)
55
56     #close main.cpp
57     main.close()
58 def copy_util_files():
59     command = "cp util/* "
60     command += folder
61     system(command)
```

```
62
63 def main():
64     verify_directory(folder)
65     verify_directory(include_directory)
66     create_includes()
67     create_main_file()
68     copy_util_files()
69
70 if __name__ == "__main__":
71     main()
```

B.1.5 Guardas Inclusão com Pragma Once

Código B.6: Script Pragma Once

```
1
2 # pragma_only.py
3 from os import mkdir
4 from os import path
5 from os import system
6
7 number_of_files = 10**4
8 number_of_includes = 3
9
10 folder =  "./pragma-only/"
11
12 include_directory = folder+"include"
13 include_path = folder+"include/{0}.hpp"
14 path_main_file = folder+"main.cpp"
15
16 content_of_include = """#pragma once
17 const int int{0} = {0};
18 """
19
20 end_of_main_file = "int main() {\n}"
21 header = """#include "{0}.h"\n"""
22
23 def verify_directory(path_name):
24     if not path.exists(path_name):
25         mkdir(path_name)
26
27 def create_include_file(path,content):
28     f =  open(path,"w+")
29     f.write (content)
30     f.close()
31
32 def create_includes():
```

```
33      #create directory
34      # create all files
35      for number in range(0,number_of_files):
36          path = include_path.format(str(number))
37          content = content_of_include.format(str(number))
38          create_include_file(path,content)
39
40  def create_main_file():
41      #open main.cpp
42      main = open(path_main_file,"w+")
43
44      #write includes 3 times
45      for number in range(0,number_of_files):
46          for x in range(0,number_of_includes):
47              content = header.format(str(number))
48              main.write(content)
49
50      #write end of file
51      main.write(end_of_main_file)
52
53      #close main.cpp
54      main.close()
55  def copy_util_files():
56      command = "cp util/* "
57      command += folder
58      system(command)
59
60
61  def main():
62      verify_directory(folder)
63      verify_directory(include_directory)
64      create_includes()
65      create_main_file()
66      copy_util_files()
67
68  if __name__ == "__main__":
69      main()
```

B.1.6 Guardas de Inclusão com Redundância

Código B.7: Script Redundância de Guardas de Inclusão

```
1
2  # redundante.py
3  from os import mkdir
4  from os import path
5  from os import system
```

```
6
7  number_of_files = 10**4
8  number_of_includes = 3
9
10 folder =  "./redundant/"
11
12 include_directory = folder+"include"
13 include_path = folder+"include/{0}.hpp"
14 path_main_file = folder+"main.cpp"
15
16 content_of_include = """#ifndef H{0}_HPP
17 #define H{0}_HPP
18 const int int{0} = {0};
19 #endif"""
20 end_of_main_file = "int main() {\n}\n"
21 header = """#ifndef H{0}_HPP
22 #define H{0}_HPP
23 #include "{0}.hpp"
24 #endif
25 """
26
27 def verify_directory(path_name):
28     if not path.exists(path_name):
29         mkdir(path_name)
30
31 def create_include_file(path,content):
32     f =  open(path,"w+")
33     f.write (content)
34     f.close()
35
36 def create_includes():
37     #create directory
38     # create all files
39     for number in range(0,number_of_files):
40         path = include_path.format(str(number))
41         content = content_of_include.format(str(number))
42         create_include_file(path,content)
43
44 def create_main_file():
45     #open main.cpp
46     main = open(path_main_file,"w+")
47
48     #write includes 3 times
49     for number in range(0,number_of_files):
50         for x in range(0,number_of_includes):
51             content = header.format(str(number))
52             main.write(content)
```

```
53          main.write("\n")
54
55      #write end of file
56      main.write(end_of_main_file)
57
58      #close main.cpp
59      main.close()
60  def copy_util_files():
61      command = "cp util/* "
62      command += folder
63      system(command)
64
65
66  def main():
67      verify_directory(folder)
68      verify_directory(include_directory)
69      create_includes()
70      create_main_file()
71      copy_util_files()
72
73  if __name__ == "__main__":
74      main()
```

APÊNDICE C – Compilação de projetos

C.1 Script de recompilações

Código C.1: Script de Recompilações

```
1
2         # *-* encoding: utf8 *-*
3         from datetime import datetime
4         from os import getcwd, chdir, makedirs, system
5         from os.path import exists
6         from yaml import load
7
8         # common variables
9         MAX_TIMES = 10
10        current_dir = getcwd()
11        output = current_dir+"/output/"
12        config_file = current_dir+"/projects.yaml"
13
14        def make_directories():
15            if(not exists(output)):
16                makedirs(output)
17
18        def write_result(message, file_name):
19            message += "\n"
20            make_directories()
21            _file = open(output+file_name, "a+")
22            _file.write(message)
23            _file.close()
24
25
26        def change_to(directory):
27            path = current_dir+"/"+directory
28            chdir(path)
29            print("Changed to {}".format(directory))
30
31
32        def compile_project(command, project_debug):
33            if not project_debug:
34                command += " > /dev/null 2> /dev/null"
35
36            system(command)
37
38
39        def clean_projects(command, project_debug):
```

```
40              if not project_debug:
41                  command += " > /dev/null 2> /dev/null"
42
43              system(command)
44
45
46          def wait_time():
47              command = "sleep 1"
48              system(command)
49
50
51          def all_project(file_name):
52              with open(file_name, 'r') as stream:
53                  content = load(stream)
54              return content
55
56
57          def set_branch(branch_name):
58              command = "git checkout {}".format(branch_name)
59              system(command)
60
61          def exec_command(command):
62              print "Using command %s"%(command)
63              system(command)
64
65
66          projects = all_project(config_file)
67
68          if not projects:
69              print("Need create a projects.yaml with projects attributes"
                    )
70
71          else:
72              for project_name, project in projects.items():
73                  print("Compiling [ {} ]".format(project_name))
74
75                  change_to(project['directory']+"/"+project['makefile'])
76
77                  for branch in project['branchs'] :
78                      set_branch(branch['name'])
79
80                      if branch['pre-command']:
81                          for command in branch['pre-command']:
82                              exec_command(command['command'])
83
84                      message = "Branch [{}][{}]:".format(branch['name'],
                            branch['description'])
```

```
85
86                      write_result(message,project_name)
87
88                      for times in range(0, MAX_TIMES):
89
90                          clean_projects(branch['clean'], project['debug'
                                ])
91                          wait_time()
92
93                          start_time = datetime.now()
94                          compile_project(branch['compile'], project['
                                debug'])
95                          end_time = datetime.now()
96
97                          elapsed = end_time - start_time
98
99                          text = "[{}/{}] : {:>4} ms".format(times+1,
                                MAX_TIMES,
100                                                             elapsed.
                                                                total_seconds
                                                                ())
101
102                         if not project['debug']:
103                             print(text)
104
105                         write_result(text, project_name)
106
107                     write_result("*"*40+"", project_name)
108
109                     if branch['pos-command']:
110                         for command in branch['pos-command']:
111                             exec_command(command['command'])
112
113             print("Finished the result was saved in output folder")
```

C.1.1 Template para compilação específica do projeto

Código C.2: Template para execução do Script de Recompilações

```
1   # project.yaml
2     <project name>:
3     directory:          <directory to project>
4     makefile:           <path to make file into directory>
5     debug:              <flag to see the project compilling>
6     branchs:
7        - name:          <branch to compile>
8          description:   <little description>
```

```
9           compile:        <line to compile the project>
10          clean:            <line to clean the object compiled>
11          pre-command:
12               - command: <command executed before compile>
13               - command: <command executed before compile>
14               - command: <command executed before compile>
15          pos-command:
16               - command: <command executed after compile>
17               - command: <command executed after compile>
18               - command: <command executed after compile>
```

APÊNDICE D – Coleta de Dados

D.1 Guardas de Inclusão

Tabela 34: Resultados Guardas de Inclusão aplicado no Linux

Tipo	Utilização de Guarda de Inclusão						
Medida	Tempos em segundos						
Amostras	**GIE**	**GII**	**PO**	**GIIPPO**	**POPGII**	**GIEPO**	**RGI**
1	1.8942	0.3631	1.2690	1.3759	1.2709	1.2828	0.1909
2	0.3203	0.2732	1.2041	1.3253	1.2378	1.2307	0.1597
3	0.2939	0.2691	1.2109	1.3102	1.2394	1.2417	0.1593
4	0.2771	0.2784	1.1840	1.3350	1.2084	1.2215	0.1581
5	0.2814	0.2752	1.1681	1.3127	1.2051	1.2428	0.1557
6	0.2817	0.2771	1.2051	1.3036	1.2358	1.2452	0.1592
7	0.2838	0.2791	1.2311	1.2976	1.2503	1.2388	0.1565
8	0.2846	0.2729	1.2077	1.3105	1.2225	1.2404	0.1559
9	0.2970	0.2797	1.2058	1.2948	1.2335	1.2658	0.1584
10	0.3108	0.2829	1.2408	1.3124	1.2077	1.2679	0.1575
Média:	0.2851	0.2851	1.2127	1.3179	1.2311	1.2477	0.1611

Tabela 35: Resultados Guardas de Inclusão aplicado no Mac OS Yosemite

Tipo	Utilização de Guarda de Inclusão						
Medida	Tempos em segundos						
Amostras	**GIE**	**GII**	**PO**	**GIIPPO**	**POPGII**	**GIEPO**	**RGI**
1	0.8753	0.8249	1.0309	0.8424	0.7564	0.9224	0.6065
2	0.8422	0.8275	0.7369	0.7748	0.7616	0.8796	0.5017
3	0.7950	0.8539	0.9389	0.8111	0.7928	0.8366	0.5277
4	0.7599	0.7930	1.1065	0.9166	0.7444	0.8158	0.5033
5	0.8539	0.7760	0.8515	1.0654	0.8573	0.7830	0.5114
6	0.9596	0.8868	1.0004	0.9729	1.3346	0.8005	0.5864
7	0.8569	0.8558	1.0348	1.3181	1.2343	0.8366	0.5791
8	0.7840	0.7600	1.0462	1.2104	1.2177	0.9315	0.5044
9	0.7987	0.8231	1.1556	1.2441	0.9490	1.2485	0.7518
10	0.9568	0.7484	1.1298	0.9836	0.8628	1.0419	0.5954
Média:	0.8482	0.8149	1.0031	1.0139	0.9511	0.9097	0.5668

Tabela 36: Resultados Guardas de Inclusão aplicado no Windows 7

Tipo	Utilização de Guarda de Inclusão						
Medida	Tempos em segundos						
Amostras	**GIE**	**GII**	**PO**	**GIIPPO**	**POPGII**	**GIEPO**	**RGI**
1	2.4736	2.4535	4.2161	4.4164	4.3262	4.3863	2.1030
2	2.3634	2.3834	4.1760	4.3162	4.3062	4.2761	2.1030
3	2.4235	2.4135	4.2962	4.2060	4.2962	4.2261	2.1331
4	2.4035	2.3934	4.4564	4.2962	4.3563	4.2060	2.1531
5	2.3634	2.3534	4.2161	4.2661	4.2962	4.3663	2.1130
6	2.3834	2.3634	4.1560	4.2561	4.2862	4.2461	2.1030
7	2.4235	2.3734	4.1660	4.2161	4.3262	4.1960	2.2032
8	2.4035	2.4235	4.1460	4.2060	4.3663	4.2261	2.2933
9	2.3934	2.4235	4.1760	4.2261	4.2862	4.2561	2.1331
10	2.3834	2.3734	4.1560	4.2060	4.3062	4.2361	2.1331
Média:	2.4015	2.3954	4.2161	4.2611	4.3152	4.2621	2.1471

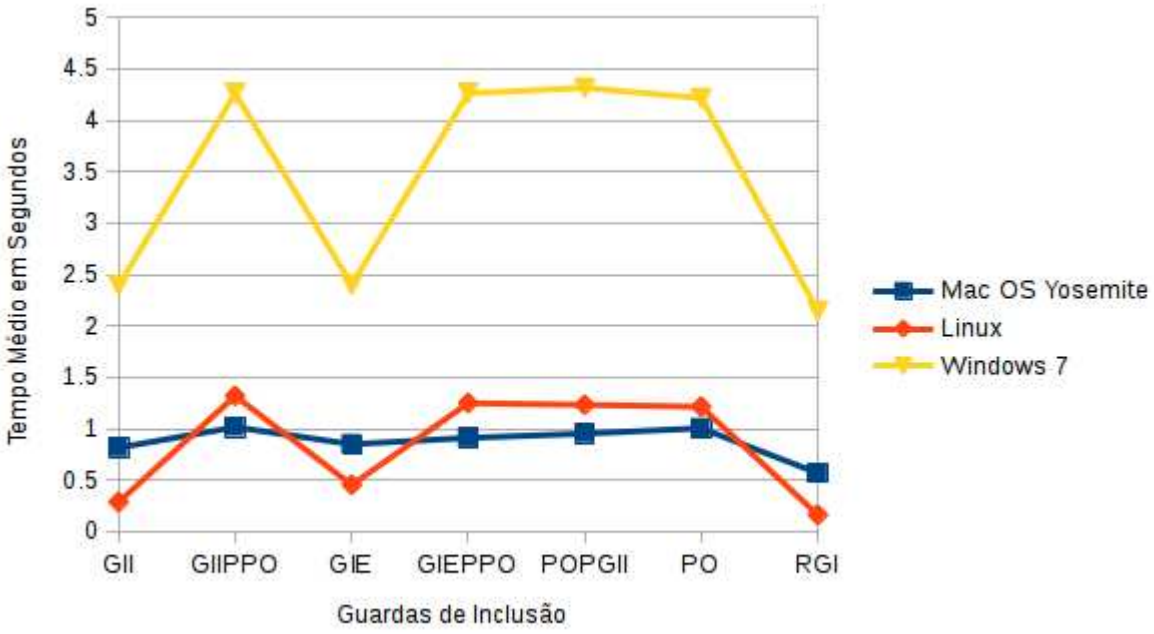

Figura 15: Dados coletados dos scripts de Guardas de Inclusão

D.2 Métodos e ferramentas aplicados a um projeto

D.2.1 Métodos que envolvem alteração de Código

Aseprite

Tabela 37: Aseprite - Alteração de Código no Linux

Aseprite	**master**	**pragma once**	**forward declaration**	**private implementation**
1	2514.9207	2418.7051	-	-
2	2333.7905	2370.1243	-	-
3	2352.6054	2358.9848	-	-
4	2341.8494	2346.8402	-	-
5	2326.8056	2347.7365	-	-
6	2342.5205	2367.7181	-	-
7	2339.0471	2347.5773	-	-
8	2335.0176	2364.5381	-	-
9	2344.1943	2352.7533	-	-
10	2347.9277	2351.9232	-	-
Média	2357.8679	2362.6901	-	-

Tabela 38: Aseprite - Alteração de Código no Mac OS Yosemite

Aseprite	**master**	**pragma once**	**forward declaration**	**private implementation**
1	788.2866	793.1793	-	-
2	767.4185	778.4403	-	-
3	764.6951	769.6590	-	-
4	764.9236	772.5233	-	-
5	779.1553	771.1125	-	-
6	763.0460	767.2010	-	-
7	764.7422	768.7916	-	-
8	768.9518	768.6012	-	-
9	764.8288	765.9876	-	-
10	765.5380	769.1677	-	-
Média	769.1586	772.4663	-	-

Tabela 39: Aseprite - Alteração de Código no Windows 7

Aseprite	**master**	**pragma once**	**forward declaration**	**private implementation**
1	2504.0313	2348.2715	-	-
2	2462.2656	2320.0910	-	-
3	2429.9063	2316.0552	-	-
4	2419.6250	2322.7848	-	-
5	2436.5938	2313.0809	-	-
6	2413.1406	2311.7142	-	-
7	2444.6719	2310.8516	-	-
8	2437.4375	2310.6613	-	-
9	2437.3125	2307.9774	-	-
10	2428.8438	2310.4009	-	-
Média	2441.3828	2317.1889	-	-

iRecoveryplusplus

Tabela 40: iRecoveryplusplus - Alteração de Código no Linux

iRecoveryplusplus	master	pragma once	forward declaration	private implementation
1	1.1434	0.8880	1.0075	0.8692
2	1.0827	0.8912	0.9056	0.8717
3	1.1257	0.8359	0.8436	0.9154
4	1.0751	0.8506	0.8626	0.8796
5	1.0623	0.8314	0.9103	0.8760
6	1.1633	0.8658	0.8285	0.8405
7	1.0963	0.8921	0.9060	0.9267
8	1.4273	0.7975	0.9121	1.0221
9	1.0871	0.8991	0.8943	0.9513
10	1.4181	0.8476	0.9039	0.9898
Média	1.1681	0.8599	0.8975	0.9142

Tabela 41: iRecoveryplusplus - Alteração de Código no Mac OS Yosemite

iRecoveryplusplus	master	pragma once	forward declaration	private implementation
1	2.7747	2.7420	2.6686	2.6628
2	2.9096	2.6853	2.6563	2.5578
3	2.8155	2.6057	2.7956	2.7908
4	2.8629	2.8325	2.6673	2.7269
5	2.8847	2.8229	2.7536	2.7051
6	2.8877	3.0290	2.7412	2.6424
7	2.8877	3.1176	2.6894	2.6555
8	2.8634	3.1709	2.6639	2.8007
9	2.7734	3.1759	2.6337	2.6727
10	2.8015	3.0473	2.6510	2.7748
Média	2.8461	2.9229	2.6921	2.6990

Tabela 42: iRecoveryplusplus - Alteração de Código no Windows 7

iRecoveryplusplus	master	pragma once	forward declaration	private implementation
1	4.4375	5.0447	4.2836	4.8445
2	4.7969	4.9847	4.2937	4.8344
3	4.7969	4.9947	4.3037	4.8344
4	4.8438	5.0047	4.2836	4.7944
5	5.1094	4.9847	4.2536	4.8344
6	5.6875	4.9947	4.3437	4.8344
7	5.0625	5.0347	4.2836	4.8244
8	5.1719	5.0648	4.3437	4.8545
9	5.0156	5.0447	4.3037	4.8545
10	4.7656	5.0347	4.3037	4.8244
Média	4.9688	5.0187	4.2997	4.8334

Pencil

Tabela 43: Pencil - Alteração de Código no Linux

Pencil	master	pragma once	forward declaration	private implementation
1	455.6515	459.8221	-	455.7540
2	458.6480	457.9709	-	453.2132
3	453.9845	458.2263	-	455.5010
4	452.3321	458.2057	-	454.1596
5	456.3933	456.9206	-	451.7834
6	452.3841	458.6638	-	451.9862
7	455.8518	461.7113	-	454.3775
8	454.1806	461.1395	-	453.9680
9	455.4113	456.5944	-	454.3351
10	455.8023	463.0633	-	453.5500
Média	455.0640	459.2318	-	453.8628

Tabela 44: Pencil - Alteração de Código no Mac OS Yosemite

Pencil	master	pragma once	forward declaration	private implementation
1	355.0881	345.0807	-	355.7133
2	352.3484	345.0180	-	354.2093
3	353.0415	346.0400	-	352.3386
4	352.9057	344.8880	-	352.3613
5	352.1713	343.9720	-	352.1329
6	352.2540	348.2278	-	352.0875
7	352.2950	345.0131	-	351.9197
8	352.9621	346.0316	-	352.4711
9	353.0637	343.7028	-	351.6541
10	354.2621	354.0800	-	352.8941
Média	353.0392	346.2054	-	352.7782

Tabela 45: Pencil - Alteração de Código no Windows 7

Pencil	master	pragma once	forward declaration	private implementation
1	538.3750	534.8552	-	520.8307
2	521.5781	525.2313	-	519.9795
3	532.3281	524.4802	-	519.9094
4	528.8750	524.3200	-	520.3501
5	533.0938	526.0625	-	524.8165
6	530.6719	530.8694	-	521.4516
7	531.1250	535.8866	-	519.6390
8	538.9375	539.3316	-	519.1383
9	528.5156	527.6548	-	518.9781
10	524.0313	526.1026	-	517.6862
Média	530.7531	529.4794	-	520.2780

Sudoku

Tabela 46: Sudoku - Alteração de Código no Linux

Sudoku	master	pragma once	forward declaration	private implementation
1	22.1422	21.8947	20.8328	21.2337
2	22.0332	22.0322	20.5527	21.2164
3	21.7976	22.1801	20.7811	21.4096
4	22.0241	22.3197	20.7530	21.4550
5	25.4559	22.2421	20.9125	21.5492
6	21.8995	22.3303	20.5124	21.5999
7	21.9687	21.7905	20.6614	21.2639
8	21.4932	22.1720	20.3108	21.3969
9	21.5503	22.1463	20.4342	21.3289
10	21.7469	22.1263	20.5389	21.4003
Média	22.2112	22.1234	20.6290	21.3854

Tabela 47: Sudoku - Alteração de Código no Mac OS Yosemite

Sudoku	master	pragma once	forward declaration	private implementation
1	38.2454	39.1759	36.0855	38.0805
2	38.3764	38.7198	36.0266	37.7686
3	38.6331	38.5172	36.1797	37.8116
4	38.4525	38.7191	36.3029	37.9859
5	38.6316	38.5743	36.5839	37.7072
6	38.2802	38.6160	36.3866	37.8561
7	38.0891	38.5843	36.3129	37.6857
8	38.0895	39.0266	36.2605	37.7838
9	38.7379	39.0707	36.1961	37.8776
10	38.6165	38.8849	36.2704	37.6040
Média	38.4152	38.7889	36.2605	37.8161

Tabela 48: Sudoku - Alteração de Código no Windows 7

Sudoku	master	pragma once	forward declaration	private implementation
1	33.1406	29.4080	27.7256	28.7771
2	31.2969	29.2778	27.8658	28.6869
3	33.1406	29.4681	28.0060	28.6469
4	32.0938	29.4681	28.5668	28.8772
5	32.7500	29.3279	28.0460	28.8071
6	32.3906	29.4380	27.8658	28.8872
7	33.4063	29.3779	27.9459	28.6268
8	33.5938	29.4180	27.9559	28.9073
9	33.2969	29.4380	27.8557	28.8372
10	32.1250	29.3579	27.8658	28.7671
Média	32.7234	29.3980	27.9699	28.7821

Qcad

Tabela 49: Qcad - Alteração de Código no Linux

Qcad	**master**	**pragma once**	**forward declaration**	**private implementation**
1	3922.6279	3841.8362	-	-
2	3924.2405	3847.9879	-	-
3	3918.2415	3842.2957	-	-
4	3927.2388	3836.0845	-	-
5	3909.7712	3829.0011	-	-
6	3922.4313	3844.4143	-	-
7	3915.6719	3842.4034	-	-
8	3934.1575	3835.4759	-	-
9	3929.0349	3841.4799	-	-
10	3927.5069	3851.2170	-	-
Média	3923.0922	3841.2196	-	-

Tabela 50: Qcad - Alteração de Código no Mac OS Yosemite

Qcad	**master**	**pragma once**	**forward declaration**	**private implementation**
1	3738.2187	3361.4522	-	-
2	3748.0250	3291.7415	-	-
3	3727.1546	3277.3348	-	-
4	3722.7333	3278.2429	-	-
5	3722.3808	3287.2635	-	-
6	3727.3305	3283.6126	-	-
7	3729.9403	3282.5581	-	-
8	3722.0891	3281.0293	-	-
9	3731.7932	3278.8484	-	-
10	3728.2715	3282.1175	-	-
Média	3729.7937	3290.4201	-	-

D.2.2 Métodos que envolvem a parametrização do *build*

Aseprite

Tabela 51: Aseprite - *Flags* de Otimização da Compilação no Linux

Aseprite	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	2428.4120	2260.5384	2514.9207	2561.8271	2319.3125	842.4427	2385.7684
2	2271.3874	2051.0073	2333.7905	2384.6942	2235.8169	694.0581	2243.8749
3	2241.6417	2050.1224	2352.6054	2374.5866	2236.5715	694.0660	2238.7497
4	2269.4441	2053.0067	2341.8494	2419.4744	2235.4428	688.8936	2232.4368
5	2270.0459	2054.9605	2326.8056	2397.2705	2236.9505	691.6836	2244.6255
6	2244.0567	2059.6010	2342.5205	2387.0151	2239.6219	686.9482	2250.8028
7	2249.4974	2053.8001	2339.0471	2402.3641	2250.0025	689.9473	2228.1283
8	2266.6575	2043.2351	2335.0176	2413.0873	2234.3189	690.0143	2230.7049
9	2240.5459	2055.2829	2344.1943	2372.7927	2232.4647	689.7057	2233.7111
10	2263.4064	2051.9990	2347.9277	2410.9841	2246.9743	688.1589	2247.4484
Média	2274.5095	2073.3554	2357.8679	2412.4096	2246.7476	705.5919	2253.6251

Tabela 52: Aseprite - *Flags* de Otimização da Compilação no Mac OS Yosemite

Aseprite	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	788.1713	609.0200	788.2866	806.1006	730.4503	224.4995	-
2	769.4564	590.3682	767.4185	798.2425	710.3456	207.2422	-
3	774.6238	590.3272	764.6951	789.8889	703.9241	207.2004	-
4	773.4311	589.9075	764.9236	795.9168	713.1742	208.3290	-
5	774.2611	593.2329	779.1553	793.6047	711.7219	207.1859	-
6	776.9802	591.1526	763.0460	798.4333	710.8471	208.0186	-
7	772.8281	592.0193	764.7422	789.6653	697.3718	206.0125	-
8	774.9873	589.5759	768.9518	793.2364	697.7304	207.3992	-
9	776.6190	587.6340	764.8288	794.2100	703.1416	208.1880	-
10	775.5665	589.8825	765.5380	794.8966	703.5831	207.0215	-
Média	775.6925	592.3120	769.1586	795.4195	708.2290	209.1097	-

Tabela 53: aseprite - *flags* de otimização da compilação no Windows 7

Aseprite	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	2285.9375	2141.9688	2504.0313	2606.3594	2248.0625	665.9063	2219.0781
2	2240.8438	2088.2188	2462.2656	2561.1094	2226.4375	611.2656	2172.0469
3	2238.7656	2094.9844	2429.9063	2544.8281	2217.8281	618.2188	2198.7031
4	2241.7813	2083.4375	2419.6250	2570.2813	2197.9531	618.3594	2173.2188
5	2264.3906	2079.0625	2436.5938	2539.1250	2223.7500	618.6875	2190.4688
6	2265.3438	2072.4063	2413.1406	2565.3438	2194.9219	620.8125	2205.1406
7	2285.8125	2093.2969	2444.6719	2549.6704	2248.1875	605.1094	2183.9844
8	2220.1406	2104.7813	2437.4375	2503.6875	2271.7500	613.6875	2160.1563
9	2225.7813	2084.2188	2437.3125	2525.3281	2260.6563	618.3750	2168.8750
10	2273.6563	2081.7500	2428.8438	2540.9219	2222.4375	617.4063	2174.0313
média	2254.2453	2092.4125	2441.3828	2550.6655	2231.1984	620.7828	2184.5703

iRecoveryplusplus

Tabela 54: iRecoveryplusplus - *Flags* de Otimização da Compilação no Linux

iRecoveryplusplus	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	1.0295	0.9205	1.1434	1.1348	0.9329	1.1792	1.0145
2	1.0814	0.9904	1.0827	1.0842	0.9084	1.1016	1.0078
3	0.9600	0.8296	1.1257	1.0619	0.8918	1.1663	0.9889
4	1.0160	0.8750	1.0751	1.2117	0.9385	1.1320	1.1048
5	1.0283	0.8781	1.0623	1.1440	0.8397	1.1591	0.9839
6	0.9734	0.8727	1.1633	1.1056	0.8799	1.1107	1.0287
7	1.0707	0.9437	1.0963	1.0816	0.9179	1.1152	0.9979
8	1.0478	0.9299	1.4273	1.1642	0.8657	1.1052	1.0088
9	1.0554	0.8168	1.0871	1.1080	0.8483	1.0804	1.0800
10	1.1668	0.8495	1.4181	1.0897	0.9279	1.1124	1.0608
Média	1.0429	0.8906	1.1681	1.1186	0.8951	1.1262	1.0276

Tabela 55: iRecoveryplusplus - *Flags* de Otimização da Compilação no Mac OS Yosemite

iRecoveryplusplus	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	3.3391	2.7805	2.7747	2.8193	2.8152	2.8780	-
2	2.8881	2.8706	2.9096	2.8318	3.1142	2.8373	-
3	2.8863	2.7529	2.8155	2.8035	2.6651	2.8276	-
4	2.7618	2.2722	2.8629	2.9031	2.6988	2.8334	-
5	2.9073	2.3670	2.8847	2.8012	3.1920	2.8224	-
6	2.8206	2.9688	2.8877	2.8034	2.7669	2.8981	-
7	2.8961	2.7507	2.8877	2.8764	2.5998	2.8582	-
8	2.8205	2.7768	2.8634	2.8438	2.6642	2.8989	-
9	2.8858	2.1318	2.7734	2.7415	2.6096	2.8480	-
10	2.7613	2.0424	2.8015	2.9054	2.5198	2.9179	-
Média	2.8967	2.5714	2.8461	2.8329	2.7646	2.8620	-

Tabela 56: iRecoveryplusplus - *flags* de otimização da compilação no Windows 7

iRecoveryplusplus	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	4.2969	5.2656	4.4375	4.9219	4.8438	4.7656	5.2813
2	4.4219	4.7031	4.7969	5.0625	4.8125	5.2188	4.9844
3	4.7500	4.8125	4.7969	5.3594	4.4375	4.4688	4.6406
4	4.6094	4.5625	4.8438	4.9375	4.3750	5.2031	4.3906
5	4.6094	4.6094	5.1094	4.9531	4.3438	5.3594	4.7031
6	4.4375	4.3438	5.6875	5.6406	4.5000	5.6406	4.8438
7	4.3750	4.8281	5.0625	5.1563	4.5156	4.7969	4.4688
8	4.5781	4.8750	5.1719	4.9531	4.3281	4.7656	5.2656
9	4.6719	4.5156	5.0156	5.4063	4.3750	4.7031	4.8125
10	5.0625	3.9688	4.7656	5.2656	4.7500	4.5313	5.1563
média	4.5813	4.6484	4.9688	5.1656	4.5281	4.9453	4.8547

Pencil

Tabela 57: Pencil - *Flags* de Otimização da Compilação no Linux

Pencil	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	455.4740	453.7972	455.6515	454.3954	461.3637	454.6087	461.1411
2	458.5687	453.0420	458.6480	455.8612	451.2379	456.3201	457.7603
3	456.1341	453.2612	453.9845	456.0386	451.0199	452.0644	452.1741
4	455.9241	453.6109	452.3321	455.1617	451.8650	458.3650	454.3638
5	455.0626	456.3227	456.3933	452.7784	451.1833	456.5019	453.1567
6	454.6253	456.1635	452.3841	458.7813	452.3104	458.1658	451.5917
7	456.1520	461.9098	455.8518	452.1305	453.0904	452.4355	455.8444
8	456.0375	451.5324	454.1806	456.6440	456.5508	455.4595	454.2039
9	457.7828	453.7152	455.4113	453.6454	451.1968	452.4086	455.3102
10	459.6924	453.2093	455.8023	455.0911	452.3534	450.9296	451.3153
Média	456.5454	454.6564	455.0640	455.0528	453.2172	454.7259	454.6861

Tabela 58: Pencil - *Flags* de Otimização da Compilação no Mac OS Yosemite

Pencil	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	355.3439	353.7290	355.0881	352.6608	349.1689	352.7384	-
2	353.1613	356.1727	352.3484	355.5543	369.1497	358.3245	-
3	352.5484	351.8661	353.0415	352.5796	375.6915	353.3902	-
4	352.7343	352.1315	352.9057	354.8242	376.2380	353.5124	-
5	353.0833	352.2203	352.1713	352.5561	378.8831	352.5611	-
6	352.6301	353.4998	352.2540	353.6754	378.1887	352.6806	-
7	353.3471	352.6812	352.2950	352.3423	357.7742	352.1714	-
8	353.5499	351.6440	352.9621	354.2137	354.8617	353.5657	-
9	352.6414	352.4142	353.0637	352.8731	356.9490	381.1381	-
10	352.0240	352.5088	354.2621	352.7468	351.7314	353.2352	-
Média	353.1064	352.8868	353.0392	353.4026	364.8636	356.3318	-

Tabela 59: Pencil - *flags* de otimização da compilação no Windows 7

Pencil	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	520.0625	500.8906	538.3750	538.0469	543.6563	543.0938	506.2969
2	523.6406	508.5938	521.5781	529.4375	524.0000	532.3906	505.2188
3	514.5313	494.6250	532.3281	533.5000	526.1094	534.9531	514.7344
4	531.6563	503.0781	528.8750	534.0938	512.1563	549.3750	512.7656
5	517.6875	505.2656	533.0938	533.9844	522.9063	547.5000	505.3750
6	511.5781	501.7969	530.6719	529.0625	509.9063	527.2344	516.5313
7	510.3594	498.9375	531.1250	532.8594	529.4063	537.8750	520.0781
8	509.7500	503.4375	538.9375	538.9063	517.6094	542.6250	520.8750
9	516.4531	501.4063	528.5156	534.2813	522.5000	549.1719	513.1875
10	511.1563	500.8125	524.0313	541.3906	520.8438	534.9688	524.3125
média	516.6875	501.8844	530.7531	534.5563	522.9094	539.9188	513.9375

Sudoku

Tabela 60: Sudoku - *Flags* de Otimização da Compilação no Linux

Sudoku	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	23.5794	22.0289	22.1422	21.9497	19.6868	21.4870	21.9286
2	22.0800	22.4456	22.0332	21.9361	19.5858	22.0021	22.0167
3	22.0984	22.0593	21.7976	21.8664	19.8344	22.0210	21.7269
4	21.7172	22.1378	22.0241	21.6025	19.6490	21.9593	21.6730
5	22.3950	22.1298	25.4559	21.9436	19.4276	21.9457	22.3169
6	22.1307	22.1869	21.8995	21.9011	19.4698	22.0589	22.0904
7	22.0020	22.1432	21.9687	22.0499	19.5573	22.2119	23.1416
8	22.1447	22.2793	21.4932	22.0821	19.5723	21.7472	22.0980
9	22.0802	22.0366	21.5503	21.8815	20.0108	21.7302	21.9612
10	22.0947	25.6994	21.7469	21.9718	19.7197	22.0414	21.8521
Média	22.2322	22.5147	22.2112	21.9185	19.6514	21.9205	22.0805

Tabela 61: Sudoku - *Flags* de Otimização da Compilação no Mac OS Yosemite

Sudoku	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	38.3382	38.8962	38.2454	38.7424	39.4471	38.4162	-
2	38.5911	38.6555	38.3764	38.2607	39.3203	38.7771	-
3	38.6160	38.9842	38.6331	38.4287	39.3242	38.0117	-
4	37.9895	38.0715	38.4525	38.0314	39.4237	38.3163	-
5	38.4379	37.9539	38.6316	38.5387	39.2862	38.7696	-
6	38.1603	38.6485	38.2802	37.8681	39.3197	38.0426	-
7	38.4832	38.0982	38.0891	38.5532	39.4502	38.4225	-
8	38.3560	38.5092	38.0895	38.0421	39.4214	37.9470	-
9	38.2962	38.8282	38.7379	38.2293	39.3191	38.2541	-
10	38.3496	38.3687	38.6165	38.5052	39.4786	38.4714	-
Média	38.3618	38.5014	38.4152	38.3200	39.3790	38.3428	-

Tabela 62: Sudoku - *flags* de otimização da compilação no Windows 7

Sudoku	**-O**	**-O0**	**-O2**	**-O3**	**-Os**	**-Ofast**	**-Og**
1	32.0469	29.0938	33.1406	35.9688	31.3281	35.5625	31.1563
2	31.8750	28.7500	31.2969	35.0156	30.3281	35.2813	29.8281
3	31.6719	29.9844	33.1406	35.0781	31.4844	34.5156	30.1094
4	30.7344	29.3594	32.0938	35.6094	31.8906	37.1563	29.9688
5	30.5000	30.2031	32.7500	36.0938	32.4688	36.4844	30.5156
6	30.7656	28.7188	32.3906	34.5156	32.2813	34.7813	29.1563
7	32.3750	28.4844	33.4063	34.2969	31.7656	35.3594	30.1563
8	29.4375	31.1094	33.5938	34.2500	30.0156	35.6406	30.3906
9	31.1563	27.4375	33.2969	37.2188	31.0781	36.6563	30.1719
10	30.9688	29.6406	32.1250	34.7188	31.4844	35.4688	29.6406
média	31.1531	29.2781	32.7234	35.2766	31.4125	35.6906	30.1094

Qcad

Tabela 63: Qcad - *Flags* de Otimização da Compilação no Linux

Qcad	-O	-O0	-O2	-O3	-Os	-Ofast	-Og
1	3910.5869	3927.3261	3922.6279	3918.3749	3890.4638	3921.3795	3951.2642
2	3920.6295	3925.3639	3924.2405	3916.5140	3884.8599	3921.8527	4004.5549
3	3918.1448	3922.5499	3918.2415	3911.9738	3881.8755	3929.4902	3926.3707
4	3919.6268	3925.5230	3927.2388	3915.1442	3887.1729	3915.2546	3929.2216
5	3913.3476	3923.8203	3909.7712	3922.1575	3885.4041	3924.2709	3924.1439
6	3922.7388	3927.4205	3922.4313	3918.6355	3888.2514	3926.2758	3928.1144
7	3915.4064	3925.0573	3915.6719	3926.8127	3885.4175	3925.8235	3934.5177
8	3938.8978	3926.8358	3934.1575	3913.8567	3888.2649	3923.7448	3922.4567
9	3917.2526	3922.5346	3929.0349	3935.2480	3887.3314	3913.7215	3928.7853
10	3915.1611	3930.4288	3927.5069	3919.6895	3886.5140	3927.3606	3924.2721
Média	3919.1792	3925.6860	3923.0922	3919.8407	3886.5556	3922.9174	3937.3701

Tabela 64: Qcad - *Flags* de Otimização da Compilação no Mac OS Yosemite

Qcad	-O	-O0	-O2	-O3	-Os	-Ofast	-Og
1	3851.4074	3109.9399	3738.2187	3748.0183	3796.1528	3802.1043	-
2	3791.8801	3078.3796	3748.0250	3735.7577	3784.8870	3786.2843	-
3	3798.0112	3082.0572	3727.1546	3737.6603	3775.5470	3780.0927	-
4	3788.9006	3068.2094	3722.7333	3735.4360	3820.6566	3779.5674	-
5	3793.7368	3076.9819	3722.3808	3735.6313	3782.0788	3778.3790	-
6	3790.2542	3076.4253	3727.3305	3738.1959	3793.0921	3782.9355	-
7	3789.6725	3079.1327	3729.9403	3738.1217	3790.3613	3808.3038	-
8	3782.4943	3073.7092	3722.0891	3738.1959	3786.5515	3789.3795	-
9	3789.4735	3072.3970	3731.7932	3735.1269	3835.3118	3776.1316	-
10	3783.6960	3074.5065	3728.2715	3736.6376	3772.8905	3782.6756	-
Média	3795.9526	3079.1739	3729.7937	3737.8782	3793.7529	3786.5854	-

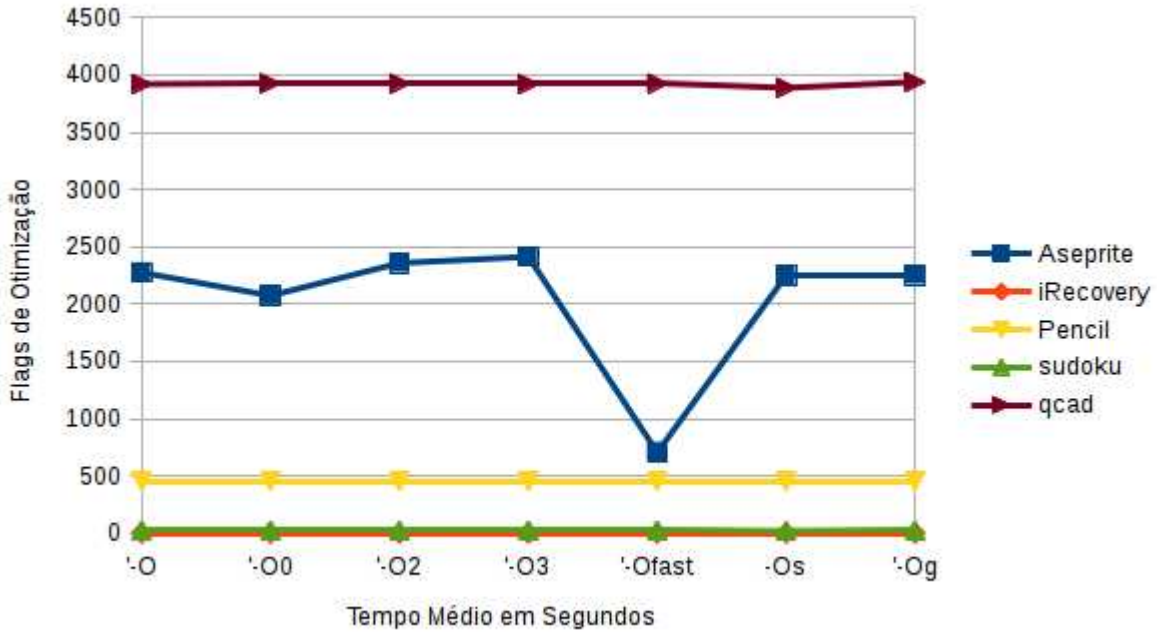

Figura 16: Dados Coletados - Utilização de *Flags* de Otimização no Linux

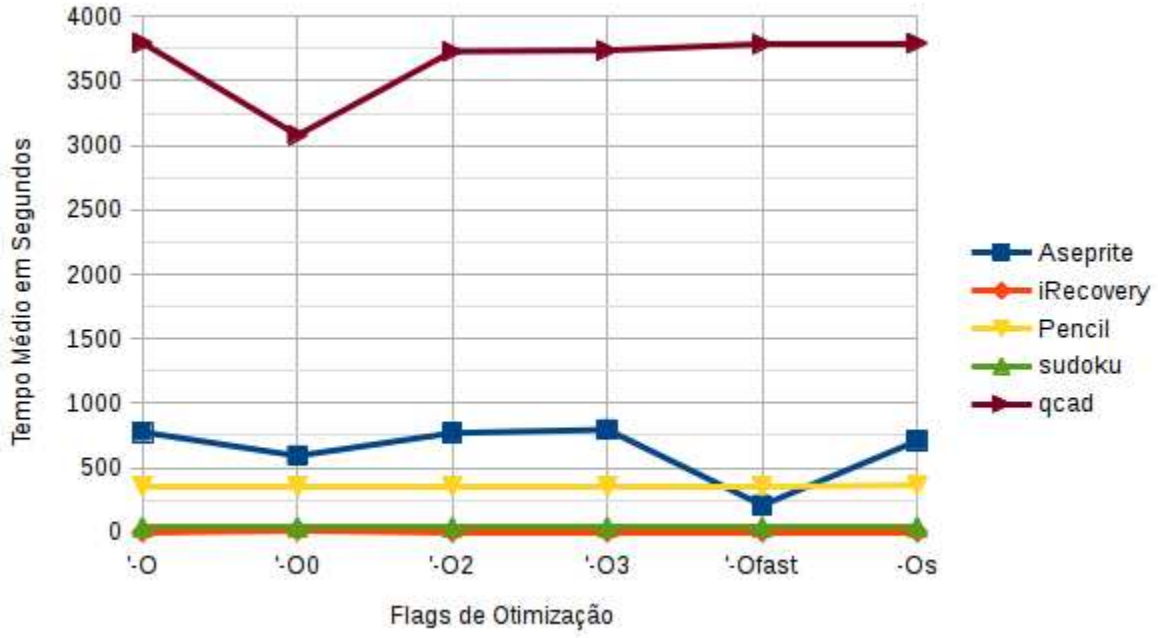

Figura 17: Dados Coletados - Utilização de *Flags* de Otimização no Mac OS Yosemite

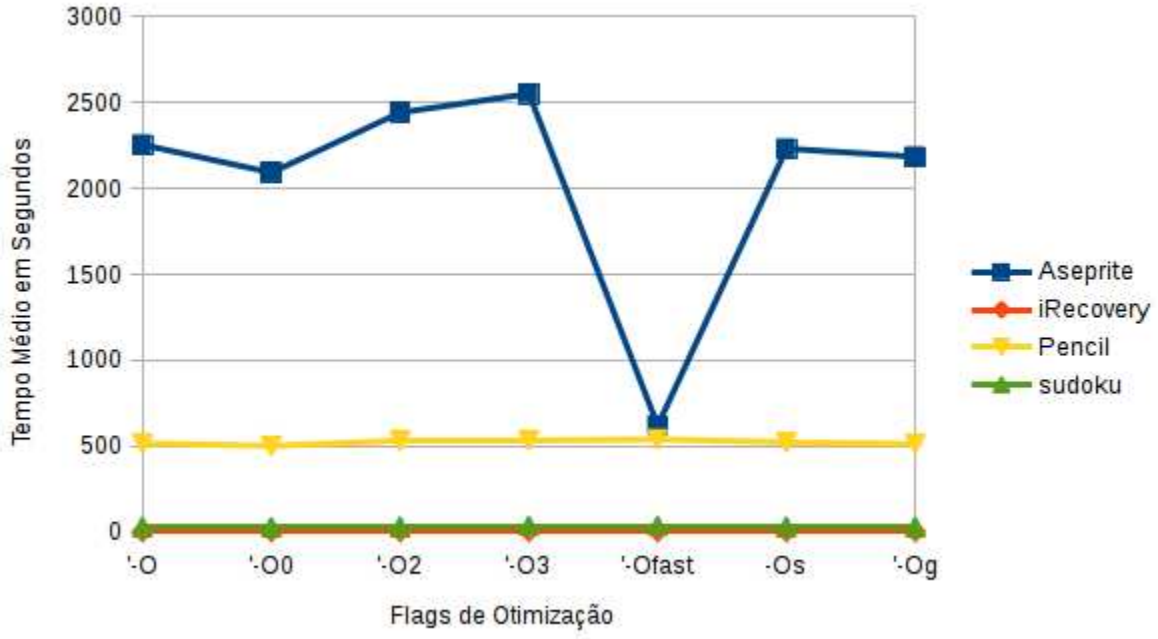

Figura 18: Dados Coletados - Utilização de *Flags* de Otimização no Windows 7

Aseprite

Tabela 65: Aseprite - *Flags* de Processamento Paralelo no Linux

Aseprite	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	1422.5747	1220.9304	1251.0829	1762.8992	1516.1982
2	1373.0582	1245.3321	1269.7347	1805.3795	1635.7142
3	1392.4281	1211.2491	1416.2245	1742.7766	1502.8607
4	1383.2098	1240.3822	1426.8681	1780.4778	1473.9050
5	1421.0623	1246.5464	1402.7551	1688.7998	1368.8088
6	1375.9400	1217.6862	1566.9732	1691.2214	1487.2643
7	1385.9952	1228.6808	1550.1486	1602.3923	1358.4499
8	1383.6518	1235.4873	1545.7303	1606.4604	1382.3776
9	1391.0771	1218.4570	1661.1944	1625.7141	1332.0942
10	1373.0357	1231.2301	1700.1015	1648.5087	1335.5501
Média	1390.2033	1229.5982	1479.0813	1695.4630	1439.3223

Tabela 66: Aseprite - *Flags* de Processamento Paralelo no Mac OS Yosemite

Aseprite	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	578.2032	575.9309	288.9686	580.5286	573.5415
2	577.5507	576.2220	284.9080	575.5498	572.1267
3	573.9103	574.6409	291.1655	577.6787	571.5593
4	580.5016	574.7476	283.9961	574.0696	570.0263
5	578.5848	584.5247	293.5214	581.9131	572.7821
6	569.2962	573.2728	287.2380	573.4792	570.0864
7	575.6596	577.6826	297.1421	577.1653	571.5139
8	576.9742	575.4852	284.0043	575.7631	575.3054
9	572.6335	572.9737	286.6796	587.5273	572.6971
10	578.8830	573.0292	288.5725	581.3931	570.0729
Média	576.2197	575.8510	288.6196	578.5068	571.9712

Tabela 67: Aseprite - *Flags* de Processamento Paralelo no Windows 7

Aseprite	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	1049.8281	1675.8594	996.0781	1006.1094	982.7031
2	1068.8594	1024.8906	1008.6094	1033.1094	1006.5469
3	1068.0000	1053.7656	1012.0156	1057.8125	995.0156
4	1055.6719	1017.4844	1042.8125	1022.9531	1022.6563
5	1893.3594	1026.8438	984.4688	1654.8281	1081.9063
6	1051.6250	1025.9063	994.5938	991.8906	1005.9063
7	1043.4844	1006.3750	984.7969	1024.9375	1083.8125
8	1026.1563	1030.8906	1021.8281	1025.7188	1012.0313
9	1054.3906	1016.9063	995.9375	987.7344	1027.6719
10	1093.0457	1052.2031	1014.5625	1027.0781	1003.6719
Média	1140.4421	1093.1125	1005.5703	1083.2172	1022.1922

iRecoveryplusplus

Tabela 68: iRecoveryplusplus - *Flags* de Processamento Paralelo no Linux

iRecoveryplusplus	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	1.0505	1.1042	1.1375	1.1768	1.2381
2	1.1425	1.0676	1.1250	1.1250	1.1480
3	1.0713	1.1384	1.1802	1.1138	1.1431
4	1.0573	1.1268	1.1377	1.1276	1.0915
5	1.0711	1.1138	1.0949	1.1724	1.0612
6	1.0971	1.1096	1.1476	1.1160	1.0889
7	1.1430	1.1175	1.0717	1.1664	1.0710
8	1.1658	1.1102	1.1188	1.0840	1.1109
9	1.1004	1.1873	1.1020	1.1301	1.0492
10	1.1769	1.1074	1.1401	1.1275	1.1678
Média	1.1076	1.1183	1.1255	1.1340	1.1170

Tabela 69: iRecoveryplusplus - *Flags* de Processamento Paralelo no Mac OS Yosemite

iRecoveryplusplus	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	2.5285	2.4667	2.4553	2.5164	2.3501
2	2.4183	2.4778	2.4793	2.8648	2.3752
3	2.4020	2.4761	2.4876	2.3739	2.3293
4	2.5213	2.4334	2.5144	2.2883	2.4144
5	2.6159	2.5339	2.4667	2.3679	2.3648
6	2.3998	2.6463	2.5849	2.3490	2.2704
7	2.4843	2.8607	2.5522	2.2977	2.3059
8	2.4747	3.3727	2.4091	2.3212	2.2715
9	2.5370	3.7488	2.4847	2.3738	2.3484
10	2.5734	2.2980	2.5224	2.3703	2.2323
Média	2.4955	2.7314	2.4957	2.4123	2.3262

Tabela 70: iRecoveryplusplus - *Flags* de Processamento Paralelo no Windows 7

iRecoveryplusplus	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	5.2500	5.4688	5.3438	5.0938	4.8438
2	5.0625	4.9688	5.0781	4.7969	4.7656
3	5.0625	4.7500	4.7969	4.7969	4.4688
4	5.6250	5.1094	5.5781	4.8125	5.0938
5	4.4531	4.7344	4.4844	4.8281	5.2969
6	4.8281	4.9688	5.3438	5.5625	5.0781
7	5.4844	5.4531	5.3438	5.5156	4.8594
8	4.7656	5.1875	5.2188	4.8125	4.6406
9	4.7188	5.0625	5.0781	5.1563	5.4531
10	4.6250	4.4844	4.8750	4.7344	4.9375
Média	4.9875	5.0188	5.1141	5.0109	4.9438

Pencil

Tabela 71: Pencil - *Flags* de Processamento Paralelo no Linux

Pencil	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	266.6753	253.7153	268.9541	276.5708	279.3204
2	262.8651	243.0800	262.3380	272.8050	280.6357
3	266.6512	243.8087	258.3085	306.8622	282.3581
4	272.8472	238.6624	255.1279	278.2493	286.3986
5	270.0952	252.4405	261.6647	278.8923	293.7820
6	268.6468	238.3732	254.2527	304.3434	295.9109
7	278.2018	239.3942	251.8955	277.3592	288.5821
8	267.8349	235.9468	255.4461	296.6876	290.8762
9	276.3785	233.7197	251.9725	279.4619	293.7961
10	287.0007	243.0883	257.5330	278.6919	294.9429
Média	271.7197	242.2229	257.7493	284.9924	288.6603

Tabela 72: Pencil - *Flags* de Processamento Paralelo no Mac OS Yosemite

Pencil	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	277.2427	271.1315	272.7361	259.9174	258.2276
2	274.8589	269.8826	286.2105	279.3511	262.6175
3	278.6200	259.0627	279.0144	273.6935	254.1095
4	282.2052	287.5182	295.9949	276.3455	268.0471
5	284.8937	274.3423	274.8261	285.8305	269.4830
6	261.9388	280.4124	280.1616	273.0806	266.0093
7	286.7162	276.5590	282.9505	291.1252	263.5765
8	274.9455	288.0063	292.4211	275.3359	273.8602
9	295.1950	281.7498	276.0287	273.9958	270.0857
10	276.5680	266.3367	280.9690	275.4276	269.8175
Média	279.3184	275.5002	282.1313	276.4103	265.5834

Tabela 73: Pencil - *Flags* de Processamento Paralelo no Windows 7

Pencil	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	261.0156	256.7500	259.6094	258.4375	283.8438
2	260.1094	256.6719	258.0625	258.2656	260.8125
3	261.8281	258.1719	262.5313	260.3594	259.9063
4	260.5469	266.9219	258.9688	258.5000	261.6250
5	262.8438	257.8750	258.7031	260.4063	259.4531
6	261.5625	272.9531	258.0469	260.0938	269.4531
7	261.3594	256.6250	257.8594	259.8906	269.3125
8	260.4375	258.4531	258.3594	259.0938	273.7656
9	273.2188	257.5000	258.1406	258.1250	265.7500
10	260.5938	257.5313	258.6875	258.3594	260.7656
Média	262.3516	259.9453	258.8969	259.1531	266.4688

Sudoku

Tabela 74: Sudoku - *Flags* de Processamento Paralelo no Linux

Sudoku	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	13.1612	13.2105	13.4355	13.5550	11.1120
2	12.8530	13.1811	13.4956	13.4440	9.7460
3	19.6240	13.4466	13.3907	13.5685	10.4685
4	13.1826	13.1441	13.2488	13.4200	10.1563
5	13.2789	13.0709	13.3169	16.7792	10.2000
6	13.0488	13.0823	13.2919	13.3107	10.1864
7	12.7417	13.2555	13.3103	13.3844	10.2037
8	13.1751	13.2540	13.3199	13.4822	9.6099
9	12.8913	13.1655	13.5218	17.4005	10.2552
10	13.0475	13.4313	13.6425	13.3221	10.7716
Média	13.7004	13.2242	13.3974	14.1667	10.2710

Tabela 75: Sudoku - *Flags* de Processamento Paralelo no Mac OS Yosemite

Sudoku	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	27.9679	29.7784	26.7719	29.8711	29.4927
2	28.6440	25.0762	28.4919	26.6717	25.7150
3	28.6159	30.4135	27.5716	29.6757	23.2773
4	26.3367	29.2411	30.0161	28.9364	26.5310
5	28.5949	28.1057	26.5836	26.9962	26.1327
6	32.7062	24.7728	26.1582	24.4893	28.0781
7	23.8390	30.8473	30.3069	29.3171	27.8168
8	26.4299	33.3270	32.5853	26.5017	31.3696
9	29.8445	25.0351	25.9885	28.6187	30.2593
10	29.7660	25.1269	25.1837	32.0158	30.7673
Média	28.2745	28.1724	27.9658	28.3094	27.9440

Tabela 76: Sudoku - *Flags* de Processamento Paralelo no Windows 7

Sudoku	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	16.6563	16.3438	16.7969	16.7344	16.0910
2	23.4844	16.4219	16.3281	16.9844	17.1236
3	16.7500	22.7656	16.7031	16.6250	17.5313
4	16.6719	16.4375	16.7500	24.7344	17.0938
5	16.4844	16.3906	16.7188	24.8906	21.5313
6	16.7500	24.3750	16.2813	16.8594	16.6250
7	16.5938	16.3281	16.3750	16.6875	17.2969
8	16.7813	16.6563	16.6719	17.5000	16.6875
9	16.5469	16.3281	16.5469	16.7031	24.4844
10	16.4844	16.3906	16.5625	16.5469	16.9531
Média	17.3203	17.8438	16.5734	18.4266	18.1418

Qcad

Tabela 77: Qcad - *Flags* de Processamento Paralelo no Linux

Qcad	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	2430.7376	2283.1428	2320.5916	1491.5863	1575.3724
2	2449.3177	2296.4186	2301.5703	1736.6482	1541.9870
3	2463.1788	2304.1666	2321.0990	1714.6174	1558.3958
4	2496.4887	2268.1321	2339.0398	1651.7457	1664.0527
5	2434.4512	2314.5346	2299.7362	1639.8338	1564.5635
6	2436.2999	2315.6281	2278.0811	1693.0271	1520.7632
7	2382.4754	2293.9582	2296.2978	1536.7922	1556.1620
8	2458.1420	2313.8770	2327.8834	1471.7916	1559.5854
9	2427.3453	2305.3579	2421.8804	1601.4796	1529.8117
10	2399.7526	2319.9531	2327.0782	1494.2895	1500.5633
Média	2437.8189	2301.5169	2323.3258	1603.1811	1557.1257

Tabela 78: Qcad - *Flags* de Processamento Paralelo no Mac OS Yosemite

Qcad	**-j 2**	**-j 4**	**-j 6**	**-j 8**	**-j10**
1	2963.7253	4299.0699	2947.3029	4446.7154	2929.5434
2	3019.4334	4105.4395	3005.8681	4115.5436	2897.6371
3	2924.2165	4269.5425	3050.5186	4196.6366	3005.6383
4	3474.6509	3803.5687	3310.4620	3504.0571	2907.6552
5	4700.4493	3663.6206	4090.5438	3026.3456	2967.7969
6	4991.6311	3549.1154	4014.6766	2990.1908	2920.4471
7	3129.8519	3491.0571	3891.0322	2921.3594	3075.4186
8	3000.6903	3154.5973	4220.4447	2961.7519	3206.6193
9	3853.9418	3220.3572	4525.4564	3024.8972	3413.1085
10	3902.5224	3298.1951	4397.8457	3038.7652	3716.1124
Média	3596.1113	3685.4563	3745.4151	3422.6263	3103.9977

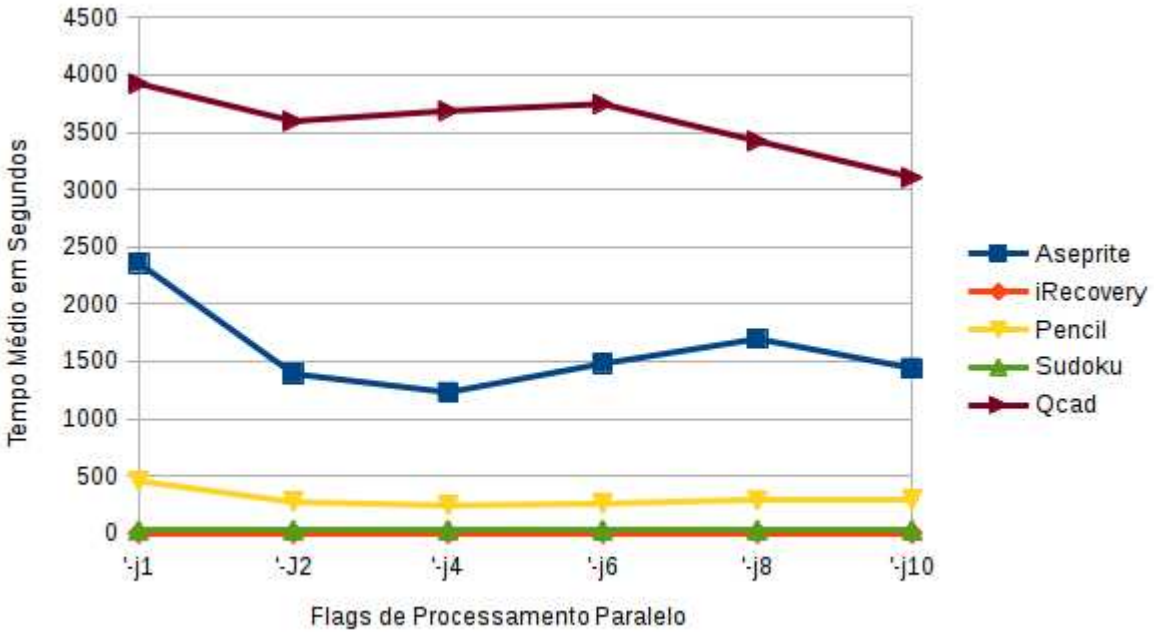

Figura 19: Gráfico *flags* de processamento paralelo no Linux

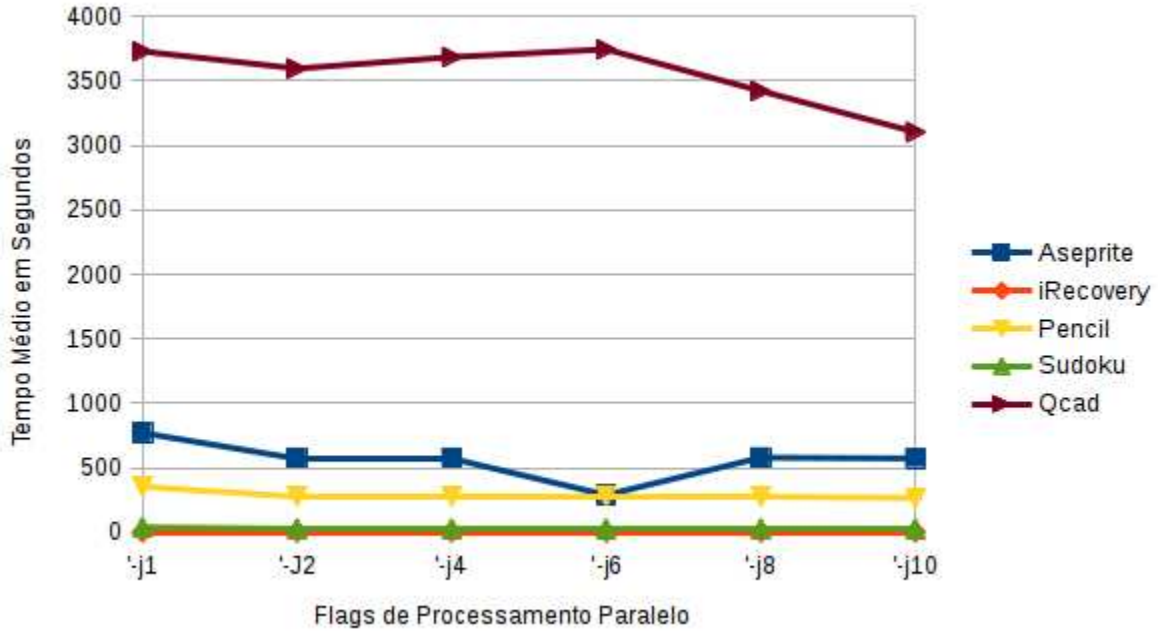

Figura 20: Gráfico *flags* de processamento paralelo Mac OS Yosemite

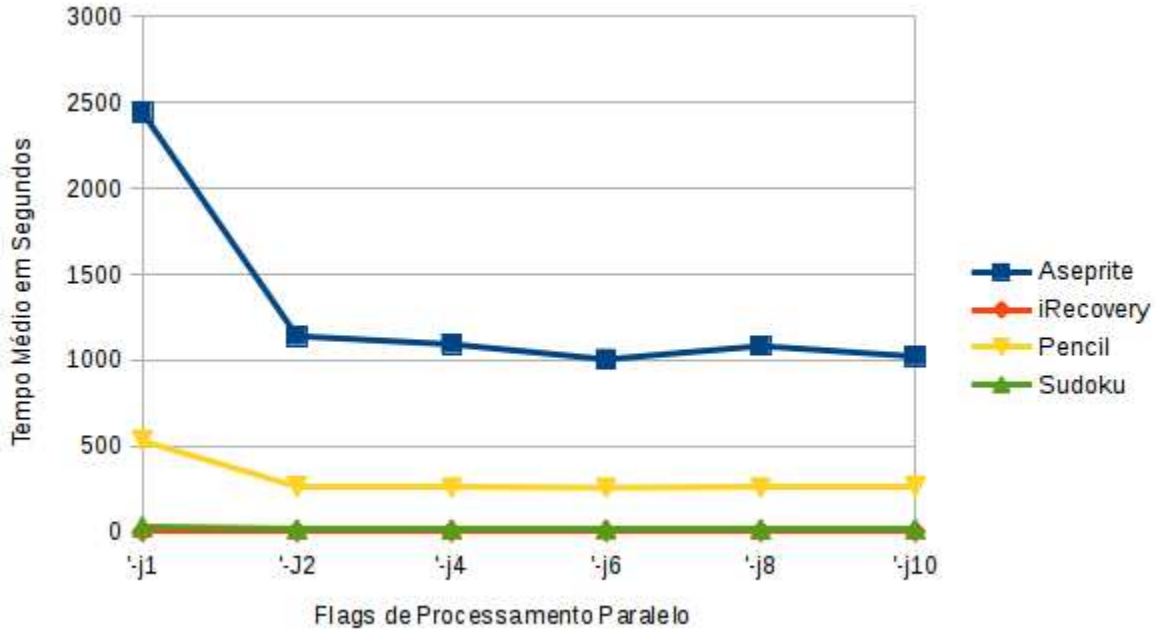

Figura 21: Gráfico *flags* de processamento paralelo no Windows 7

D.2.3 Métodos que envolvem o uso de ferramentas externas

Aseprite

Tabela 79: Aseprite - Ferramentas Auxiliares no Linux

Aseprite	**ccache**	**gold**
1	2416.7725	2299.0718
2	327.7167	2210.9459
3	324.1109	2350.9201
4	328.9366	2308.0564
5	345.3292	2219.1003
6	347.0681	2288.3468
7	332.0379	2314.2384
8	329.9954	2280.3554
9	328.3070	2229.5886
10	325.5209	2206.4399
Média	540.5795	2270.7064

Tabela 80: Aseprite - Ferramentas Auxiliares no Mac OS Yosemite

Aseprite	**ccache**	**gold**
1	911.7870	-
2	144.8385	-
3	145.3597	-
4	148.3864	-
5	133.6848	-
6	130.1691	-
7	127.6144	-
8	120.7987	-
9	117.4732	-
10	118.0802	-
Média	209.8192	-

Tabela 81: Aseprite - Ferramentas Auxiliares no Windows 7

Aseprite	**ccache**	**gold**
1	2365.1563	-
2	596.2813	-
3	586.1250	-
4	605.1875	-
5	572.9375	-
6	587.1563	-
7	584.3125	-
8	571.2500	-
9	573.7656	-
10	575.1094	-
Média	761.7281	-

iRecoveryplusplus

Tabela 82: iRecoveryplusplus - Ferramentas Auxiliares no Linux

iRecoveryplusplus	**ccache**	**gold**
1	1.1174	0.9923
2	1.1672	0.9687
3	1.1334	1.0159
4	1.2546	1.0125
5	1.1018	1.0264
6	1.0957	1.0106
7	1.1110	0.9764
8	1.1503	0.9424
9	1.1274	1.0066
10	1.1764	1.0576
Média	1.1435	1.0009

Tabela 83: iRecoveryplusplus - Ferramentas Auxiliares no Mac OS Yosemite

iRecoveryplusplus	**ccache**	**gold**
1	3.5136	-
2	2.5199	-
3	2.5235	-
4	2.5468	-
5	2.5058	-
6	2.6408	-
7	2.4701	-
8	2.5113	-
9	2.5524	-
10	2.5642	-
Média	2.6348	-

Tabela 84: iRecoveryplusplus - Ferramentas Auxiliares no Windows 7

iRecoveryplusplus	**ccache**	**gold**
1	5.1875	-
2	4.8438	-
3	5.3125	-
4	5.2344	-
5	5.0469	-
6	5.0625	-
7	5.2500	-
8	4.6719	-
9	4.6406	-
10	5.1563	-
Média	5.0406	-

Pencil

Tabela 85: Pencil - Ferramentas Auxiliares no Linux

Pencil	ccache	gold
1	474.0012	459.5160
2	137.0085	458.7100
3	138.0875	457.7532
4	140.2100	456.7455
5	141.4131	458.2821
6	139.4786	456.1067
7	140.0177	458.7202
8	140.1902	455.9186
9	140.1566	456.5629
10	140.0372	455.3661
Média	173.0601	457.3681

Tabela 86: Pencil - Ferramentas Auxiliares no Mac OS Yosemite

Pencil	ccache	gold
1	378.6207	-
2	74.1466	-
3	75.3693	-
4	73.7821	-
5	73.5666	-
6	73.7731	-
7	73.3548	-
8	73.2855	-
9	73.8306	-
10	74.1168	-
Média	104.3846	-

Tabela 87: Pencil - Ferramentas Auxiliares no Windows 7

Pencil	ccache	gold
1	615.7969	-
2	135.4063	-
3	136.8594	-
4	134.1406	-
5	132.4688	-
6	131.5625	-
7	136.4844	-
8	132.2031	-
9	129.3750	-
10	132.5625	-
Média	181.6859	-

Sudoku

Tabela 88: Sudoku - Ferramentas Auxiliares no Linux

Sudoku	**ccache**	**gold**
1	23.9157	25.3930
2	2.2334	20.7669
3	2.1856	20.4777
4	2.1651	20.8600
5	2.2444	20.7520
6	2.3041	20.6778
7	2.2018	20.7493
8	2.1199	20.6848
9	2.1500	20.6467
10	2.0455	20.6966
Média	4.3565	21.1705

Tabela 89: Sudoku - Ferramentas Auxiliares no Mac OS Yosemite

Sudoku	**ccache**	**gold**
1	44.0081	-
2	3.2837	-
3	3.2250	-
4	3.2143	-
5	3.1913	-
6	3.1352	-
7	3.1359	-
8	3.1883	-
9	3.2088	-
10	3.1640	-
Média	7.2755	-

Tabela 90: Sudoku - Ferramentas Auxiliares no Windows 7

Sudoku	**ccache**	**gold**
1	40.4219	-
2	5.1250	-
3	5.0156	-
4	4.7656	-
5	5.2031	-
6	5.0781	-
7	4.9688	-
8	4.9219	-
9	4.8281	-
10	5.1875	-
Média	8.5516	-

Qcad

Tabela 91: Qcad - Ferramentas Auxiliares no Linux

Qcad	ccache	gold
1	4053.0217	3864.8533
2	1575.0124	3842.8008
3	1575.8025	3844.9988
4	1586.2610	3853.5289
5	1573.3699	3846.0364
6	1573.6939	3847.6719
7	1571.6429	3841.4281
8	1577.4421	3842.1131
9	1573.8719	3846.6612
10	1575.6482	3843.5791
Média	1823.5766	3847.3672

Tabela 92: Qcad - Ferramentas Auxiliares no Mac OS Yosemite

Qcad	ccache	gold
1	4429.7385	-
2	297.5427	-
3	210.7286	-
4	213.6511	-
5	221.6690	-
6	231.8735	-
7	234.9510	-
8	239.9830	-
9	266.4890	-
10	296.9332	-
Média	664.3560	-

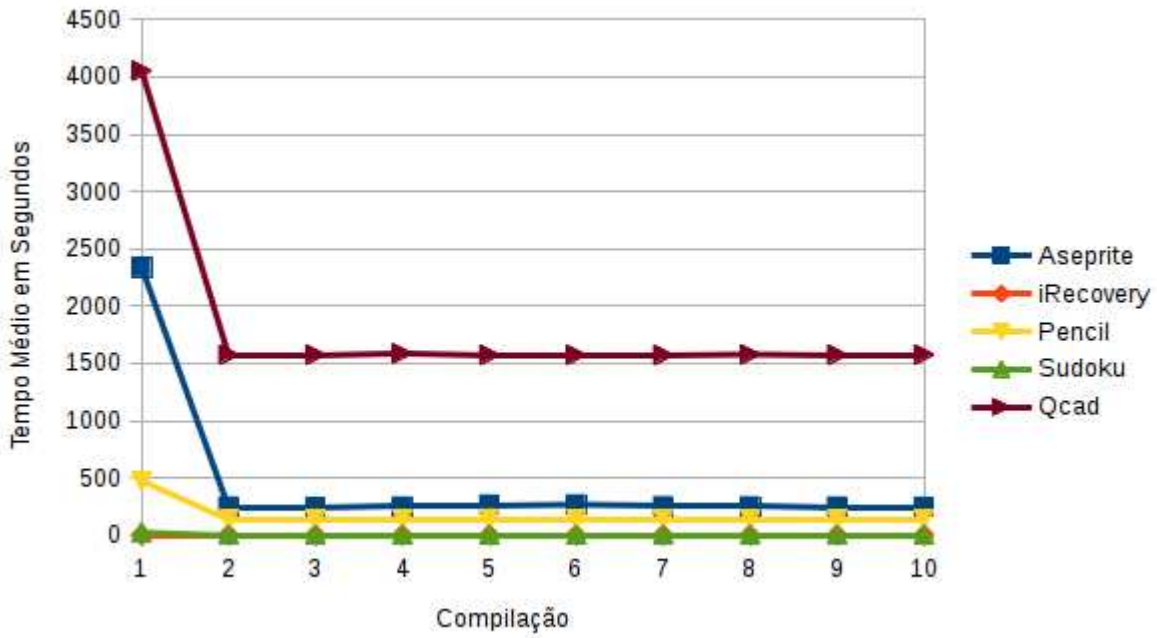

Figura 22: Gráfico dados ferramenta ccache no Linux

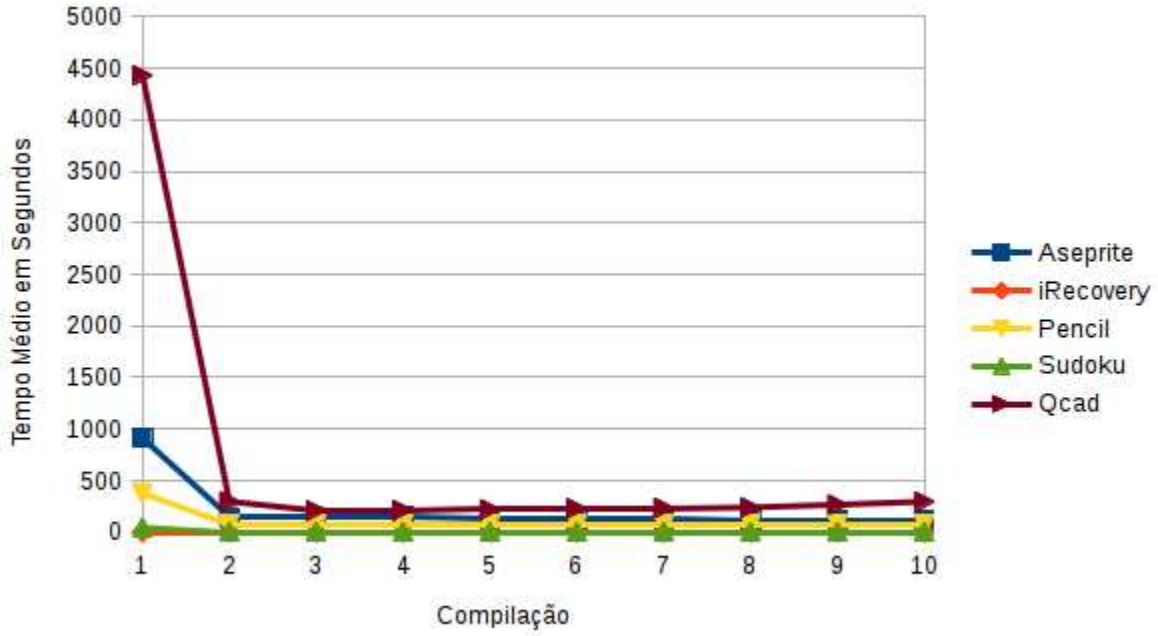

Figura 23: Gráfico dados ferramenta ccache no Mac OS Yosemite

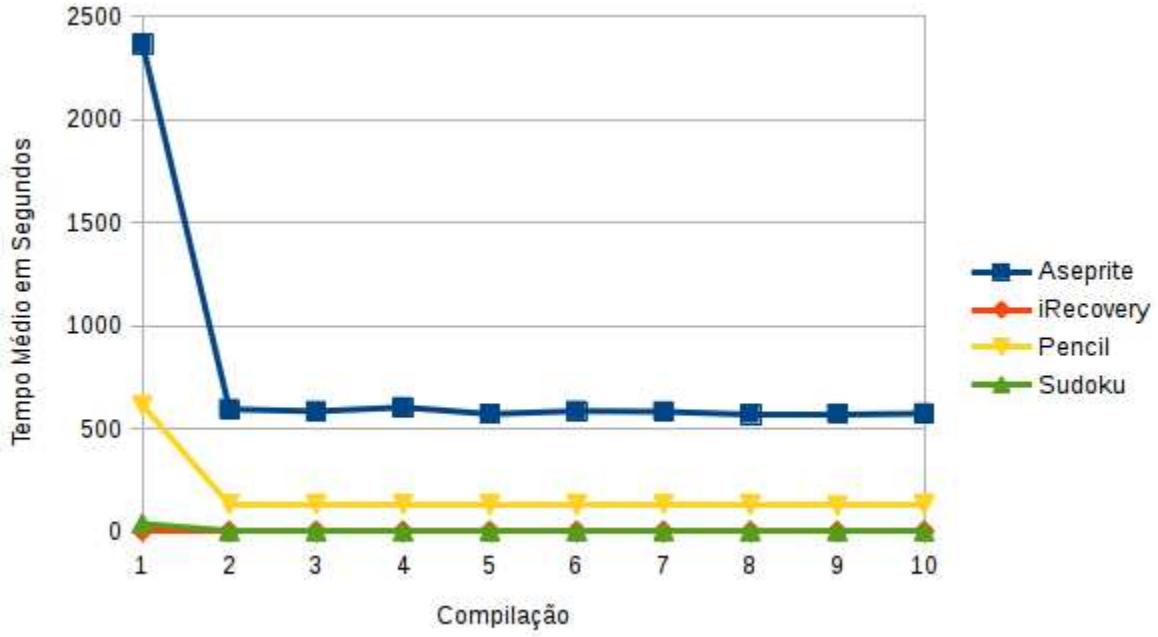

Figura 24: Gráfico dados ferramenta ccache no Windows 7

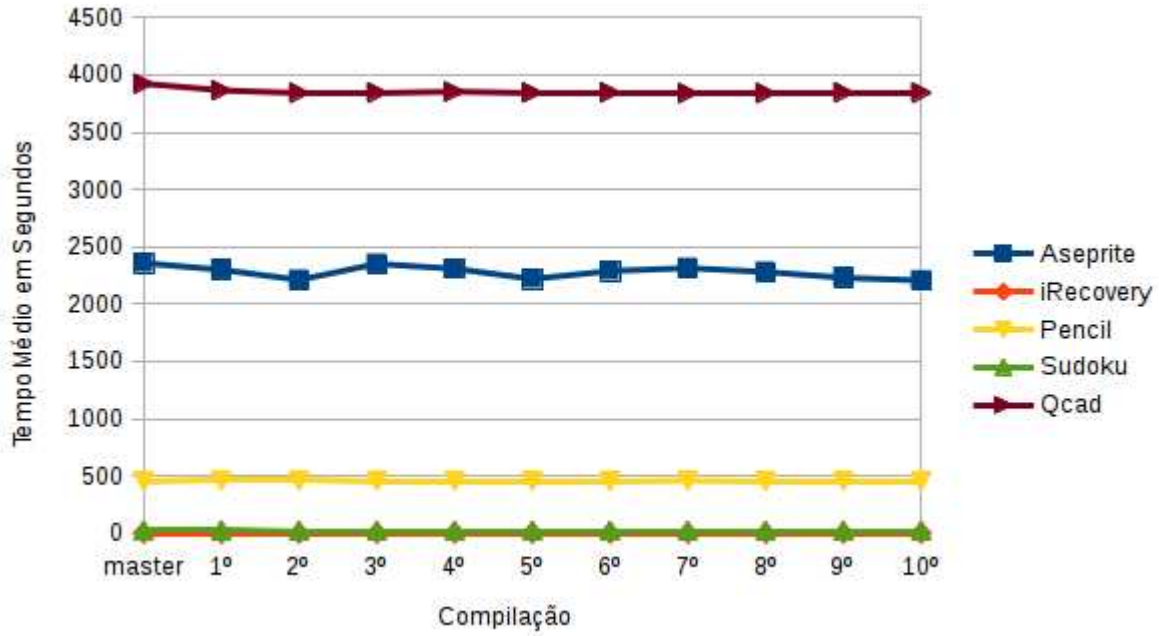

Figura 25: Gráfico dados ferramenta gold no Linux